**Compaq Computer Corporation**
**Phoenix Technologies Ltd.**
**Intel Corporation**

# Plug and Play BIOS Specification

**Version 1.0A**

**May 5, 1994**



# Table Of Contents

# References

Plug and Play ISA Specification Version 1.0A May 5, 1994
Send email to plugplay@microsoft.com to obtain a copy.

EISA Specification Version 3.12
Contact BCPR Services Inc to obtain a copy.

Extended System Configuration Data Specification Version 1.02a
Contact Intel Corporation to obtain a copy.

Device Identifier Reference Table & Device Type Code Table
Browse the PlugPlay forum on CompuServe to obtain a copy.

# 1.0 Overview

This Plug and Play BIOS Specification defines new functionality to be provided in a PC compatible system BIOS to fulfill the goals of Plug and Play. To achieve these goals, several new components have been added to the System BIOS. Two key areas that are addressed by the System BIOS are resource management and runtime configuration.
Resource management provides the ability to manage the fundamental system resources which include DMA, Interrupt Request Lines (IRQs), I/O and Memory addresses. These resources, termed ***system resources,*** are in high demand and commonly are over-allocated or allocated in a conflicting manner in ISA systems, leading to bootstrap and system configuration failures. A plug and play system BIOS will play a vital role in helping to manage these resources and ensure a successful launch of the operating system.
In its role as resource manager, a Plug and Play BIOS takes on the responsibility for configuring Plug and Play cards, as well as systemboard devices during the power-up phase. After the POST process is complete, control of the Plug and Play device configuration passes from the system BIOS to the system software. The BIOS does, however, provide configuration services for systemboard devices even after the POST process is complete. These services are known as Runtime Services.
Runtime configuration is a concept that has not previously existed in a System BIOS before. The system BIOS has not previously provided the ability to dynamically change the resources allocated to systemboard devices after the operating system has been loaded. The Plug and Play BIOS Specification provides a mechanism whereby a Plug and Play operating system may perform this resource allocation dynamically at runtime. The operating system may directly manipulate the configuration of devices which have traditionally been considered static via a System BIOS device node structure.
In addition, a Plug and Play System BIOS may also support event management. By means of the interfaces outlined in this document, the System BIOS may communicate the insertion and removal of newly installed devices which have been added to the system at runtime. The event management support defined by this specification are specific to devices controlled by the system BIOS, such as docking a notebook system to, or undocking it from, an expansion base. This event management does not encompass the insertion and removal of devices on the various expansion busses.
This document describes the BIOS support necessary for both systemboards and add-in boards with Option ROMs.

## 1.1 Goals of a Plug and Play System BIOS

Considering the scope of Plug and Play, the following are the goals of the Plug and Play BIOS Specification.
**Maximize ISA compatibility**

This is the key consideration in a system BIOS. It is considered unacceptable to change the architecture of a System BIOS to prevent the thousands of ISA cards and software programs that rely on the system BIOS for services.

**Eliminate resource conflicts during the POST procedure**

A common problem that plagues many ISA systems today is the fact that there are a lot more devices available than there are system resources. In this environment, devices are bound to have conflicting resources. The system BIOS will now play a key role to help prevent these resource conflicts by not enabling devices which conflict with the primary boot devices, and relocating boot devices, if necessary, to allow a successful load of the operating system. It is the role of the operating system to provide support for communicating irreconcilable resource conflicts to the user.

**Support Plug and Play ISA cards**

A Plug and Play system BIOS is responsible for the isolation, enumeration, and optional configuration of Plug and Play ISA cards. These cards, which provide information on their resource requirements and permit software to configure those resources, will allow the system BIOS to arrive at a conflict free configuration necessary to load the operating system.

**Allow dynamic configuration of systemboard devices**

Systemboard devices have traditionally been treated as having somewhat static configurations. It is a goal of the Plug and Play BIOS specification to provide a standard mechanism whereby systemboard devices may be configured dynamically by system software. This will grant configuration management software a great deal of flexibility when system resources are in demand and alternate configurations are necessary.

*Note: Dynamic device configuration requires explicit device driver support.*

**Provide system event notification**

The system BIOS is capable of detecting certain hardware events that could affect the system configuration. By providing an event notification mechanism, an operating system can recognize the event and process any necessary configuration changes.

**Hardware and Operating System independence**

The extensions to the system BIOS isolate the systemboard hardware through well defined interfaces and structures. The system device nodes represent devices that are controlled by the system BIOS. The operating system requires no specific knowledge of the systemboard in order to control these devices, and instead relies on the system BIOS to isolate it from the underlying hardware.

## 1.2 Enhancements to the current BIOS architecture

The Plug and Play BIOS Specification attempts to make several improvements to the current PC system BIOS architecture to achieve the goals stated previously.

- **Perform resource allocation and conflict resolution at POST time.**

  The current System BIOS Architecture performs no such resource management at POST time. The goal is to increase the probability of successfully bootstrapping into the OS by specifying resource management at POST time.

- **Actively monitor the INT 19h bootstrap vector**

  The current System BIOS Architecture allows option ROMs to hook INT 19h indiscriminately. By actively monitoring control of INT 19h, the System BIOS may regain control of the Bootstrap process to ensure that the Operating System is loaded from the proper device and in the proper manner.

- **Provide a mechanism for Remote Program Load**

  The current architecture provides no specific support for RPL. Consequently, RPL devices must resort to hooking the INT 19h bootstrap vector or INT 18h, the alternate bootstrap vector. Hooking these vectors can interfere with system specific security features, as well as result in bootstrap failures. The method and support for booting from RPL devices is beyond the scope of the Plug and Play BIOS Specification. A separate specification should define explicit support for RPL devices.

- **Provide Runtime Configuration Support**

  Proprietary techniques exist to support device resource configuration and reporting. The Plug and Play BIOS Specification defines specific, standard interfaces whereby configuration software may identify and configure devices on the systemboard.

- **Provide Dynamic Event Notification**

  A further extension of the Runtime Configuration Support allows the System BIOS to report dynamic configuration events to the Plug and Play software such that new devices added into the system may be resource managed. This dynamic event notification interface is specific to devices controlled by the system BIOS. It does not encompass the insertion and removal of devices on the various expansion busses

# 1.3 Elements of the Plug and Play BIOS architecture

## 1.3.1 Bi-modal functionality

All Plug and Play BIOS Services which are accessible at runtime support a bi-modal interface. The two modes supported are 16-bit Real Mode and 16-bit Protected Mode. These two modes are sufficient to support a wide variety of operating environments. Real Mode interfaces are defined in terms of the segment and offset of the service entry point.
Protected Mode interfaces specify the code segment base address so that the caller can construct the descriptor from the segment base address before calling the interface from protected mode. The offset value is the offset of the entry point. It is assumed that the 16-Bit Protected Mode interface is sufficient for 32-Bit Protected Mode callers. However, it is important to note that Plug and Play BIOS functions will access arguments on the stack as a 16-bit stack frame. Therefore, the caller must ensure that the function arguments are pushed onto the stack as 16-bit values and not 32-bit values. For function arguments that are pointers, the pointer offset and data should be contained within the first 64K bytes of the segment. Refer to section 4.4 Plug and Play Installation Check for a complete description of the bi-modal interface.

## 1.3.2 OS Independence

The Plug and Play BIOS services, which are accessible during normal system operation, are defined in a manner independent from the operating system. The BIOS System Device Nodes are a compact form of a device node tailored specifically to the configuration of systemboard devices.
A Plug and Play OS which complies with the general framework of the Plug and Play Architecture requires a software isolation/translation layer between the System BIOS and the OS.
The isolation/translation software performs the task of translating the generic BIOS interfaces defined in this specification into those required to support configuration management in the desired operating environment.

## 1.3.3 Expandability

Throughout the Plug and Play BIOS Specification care was taken to provide a mechanism for extensibility of this specification. All significant structures and interfaces are defined with revision identifiers. These revision identifiers provide a mechanism whereby the interfaces defined may be extended so long as the interfaces remain backward compatible to the original specification.

# 1.4 Installation Structure

Section 4.4 of this specification defines the Plug and Play installation check procedure and structure. This mechanism defines a structure which may be located on any 16-byte boundary within the System BIOS address space of 0F0000h - 0FFFFFh. Software which must determine if it is operating on a platform supporting a Plug and Play BIOS, should scan the specified address space searching for the ASCII string "$PnP" on 16-byte boundaries. If the software identifies such a string on a 16-byte boundary, it must validate that it has indeed found a Plug and Play Installation Check Structure by verifying the structure's checksum and validate either the version field or the length field or both. A valid checksum indicates that the system BIOS provides all of the required functions of the Plug and Play System BIOS specification. Specifying this structure in this manner permits it to float anywhere in the specified address range. This permits the System BIOS developer to locate the structure within their ROM without having to be concerned about it interfering with other structures that they may have specified at fixed addresses.

# 2.0 System BIOS Initialization

The Power On Self Test (POST) procedure of a system BIOS is designed to identify, test, and configure the system in preparation for starting the operating system. At the completion of POST, the PC compatible system BIOS attempts to have all of the appropriate devices enabled in order for them to be properly recognized and functioning when the operating system loads.

Over the years, PC compatible systems have become much more sophisticated in terms of the bus architectures supported and the devices attached. As these PC compatible systems have evolved and become more sophisticated, so has the system BIOS, which is responsible for the initial configuration of these devices. However, one component has remained relatively constant in a PC compatible system. This is the ***system resources***. ***System resources,*** as described in this document include DMA channels, Interrupt Request Lines (IRQs), I/O addresses, and memory.

As the sophistication of these systems increases with more and more devices, the possibility of resource conflicts also increase, leading to a possible boot or system failure. The Plug and Play BIOS specification is defined to solve the problems that occur with resource conflicts. Specifically, the Plug and Play BIOS is taking on a new responsibility to ensure that the operating system is loaded with a conflict free set of resources, as well as indicating to the operating system the resources that are currently used by systemboard devices.

## 2.1 System BIOS POST Requirements

In order to achieve the goals of Plug and Play, the system BIOS POST is responsible for achieving the requirements listed below:

- **Maintain ISA POST compatibility**
  The important issue of this broad requirement is that a Plug and Play system BIOS is responsible for the same POST requirements of an existing PC compatible system BIOS. This document focuses only on the enhancements necessary to a PC compatible system BIOS and assumes that the basic BIOS POST initialization is still performed.

- **Configuration of all static devices known to system BIOS**
  At a minimum, this includes system board devices. It can also include Plug and Play ISA Cards and devices located on EISA, ISA, PCI, or any of the other static bus architectures available. How this configuration is completed will be described in a later section.
- **BIOS POST Resource arbitration**
  The system BIOS must now be aware of ***system resource*** usage. Using the information provided through runtime services (described in a later section), along with resource information known to the system BIOS, critical resource conflicts can be avoided. Loading the operating system with a conflicting device disabled is better than causing a resource conflict and a possible system failure.
- **Initialization of the Initial Program Load (IPL) device**
  It is the responsibility of the system BIOS POST to make sure that resources for the IPL device get allocated correctly in anticipation of a successful load of the operating system. If “disabled” IPL devices are needed to achieve boot, then the system BIOS POST should take the initiative to reenable "disabled" IPL devices in an intelligent sequence to provide the best opportunity for system boot.
- **Support for both Plug and Play and Non-Plug and Play Operating Systems**
  The Plug and Play system BIOS POST must configure the system to operate with both Plug and Play aware, as well as non-Plug and Play operating system. In non-Plug and Play environments, either the system BIOS or the appropriate system software (device drivers) must configure all devices (Plug and Play ISA Cards, PCI devices, etc.). This will allow all environments to load exactly as they would on a standard PC compatible systems. However, in a Plug and Play environment, the system BIOS can now assist the operating system to perform features such as runtime configuration of system board devices and event recognition when system board devices have changed.

### 2.1.1 System Board Storage Requirements

Adding optional static resource allocation capabilities to the Plug and Play BIOS POST procedure will require additional storage. This storage is necessary for maintaining information about ***system resources*** that have been explicitly assigned to the boot devices as well as the system resources being utilized by ISA devices in the system. The amount of storage that will be necessary is platform specific, but could exceed the amount of storage normally available in PC compatible systems.
If the static resource allocation option is implemented, then the system BIOS is required to follow the function interface defined later in this document. This interface provides the mechanism for system software to specify the information about these system resources. How the information is actually stored in the nonvolatile storage on the system is left up to the BIOS implementor.
This new storage must be readily available and dependable during the system BIOS POST for the system BIOS to provide effective resource allocation. The type of storage, which can be either Flash, CMOS, NVRAM, or some other type of nonvolatile storage, and the amount of additional storage needed will vary depending on the systemboard requirements. It is left to the systemboard manufacturer to make available additional storage to the system BIOS, and the BIOS suppliers responsibility to manage and allocate this nonvolatile storage.

### 2.1.2 System BIOS Resource Management

A key element of a Plug and Play BIOS is to provide accurate resource management. Management of ***system resources***, which includes DMA, IRQs, I/O, and Memory, is vital to a system BIOS POST if it is to guarantee successful loading of the operating system. Unfortunately, there is no clear defined procedure for how these ***system resources*** should be allocated by the system BIOS. This section will describe how the system BIOS POST can manage resources and will outline the different methods that can be used to allocate the system resources.

### 2.1.3 Isolating Committed Resources

The first step to resource management is to determine ***system resources*** that are statically allocated to devices in the system. These resources can be located on ISA cards, systemboard devices, or any other device present in the system. Unfortunately, it is very difficult, if not impossible, to accurately determine the resources used, unless these devices provide information about the system resources they will use. With this in mind, it is necessary for an external program to help isolate the resources that these devices are using. How this external program determines the resources consumed by these devices is beyond the scope of this document. However, what is within the scope is the interface that the system BIOS provides to indicate resources that are allocated to the ISA devices.
Function 09h, ***Set Statically Allocated Resource Information,*** of the runtime services is designed to support an external program that can indicate the resources that are allocated to the static ISA devices in the system. Through this interface, the ***system resources*** utilized by these ISA devices will be saved in nonvolatile storage. This will allow the system BIOS to ensure the configuration of the boot devices in the system do not conflict with any static ISA devices during the POST configuration process.

### 2.1.4 System BIOS Resource Allocation

There are three fundamental methods that the system BIOS POST can use to allocate resources to devices. They are:
*Static Resource Allocation* - Allocate resources based on the last working configuration of the system. This requires that the resources assigned to specific devices in the system be saved in nonvolatile storage on the system. This configuration information must be accessible to the system BIOS during POST. The interface and format for storing the resource information explicitly assigned to every device in the system may be stored in an OEM specific format or it may follow the *Extended System Configuration Data (ESCD)* format. Refer to the *ESCD Specification* for a complete description of the ESCD and its interfaces. The ESCD interface provides a mechanism for allowing system software the ability to lock the system resources allocated to specific devices in the system. This will allow the configuration of devices to remain consistent between operating sessions.
*Dynamic Resource Allocation* - Dynamically auto-configure the systemboard and Plug and Play devices in the system. At a minimum, the system BIOS must ensure that only the primary boot devices are properly configured to boot the system software. When loaded, the system software is responsible for dynamically configuring all remaining devices. Depending on the system's architecture, the BIOS may have to implement Function 09h, ***Set Statically Allocated Resource Information,*** to guarantee a conflict free boot device configuration.
*Combined Static and Dynamic Resource Allocation* - Allocating resources based on the configuration information specified for the last working configuration for the system, as well as dynamically configuring the Plug and Play devices in the system, which were not specified in the last working configuration resource information.
A system BIOS can use any one of these methods for allocating system resources to the devices in the system. What is important for each of these methods to work successfully is an accurate accounting of the committed resources used in the system. It is important to note that the primary responsibility for system BIOS resource allocation is to ensure that the primary boot devices are configured properly to boot the Plug and Play operating system. If the Plug and Play system BIOS chooses to only configure the primary boot devices, the appropriate system software or Plug and Play operating system will be responsible for configuring any unconfigured devices.

#### Static Resource Allocation

This method assumes that the system software has specified the appropriate resource configuration information to the system BIOS for ALL devices in the system. As mentioned above, it is an option that the system BIOS interface, for allowing system software to provide the last working configuration information to the system BIOS, follows the *Extended System Configuration Data (ESCD)* format.
Once this information has been saved by the system BIOS, this information is used by the BIOS during POST to allocate resources to all of the configurable devices that are known to the system BIOS.

There are certain benefits that can be realized by supporting Static Resource Allocation. First, the configuration of every device in the system is saved in nonvolatile storage which allows the BIOS to allocate the appropriate resources to the devices in the system during POST. This allows the last working configuration to be maintained from boot to boot. Another benefit comes from the ability to explicitly assign, or lock, the resources allocated to any Plug and Play card in the system. Static resource allocation will require nonvolatile storage on the system for storing the resource allocation for each device in the system.
*Example:* The EISA architecture is an example of an architecture which uses static resource allocation. The EISA configuration utility is responsible for determining device resource allocations, then storing that information for the BIOS. Upon initialization, the system BIOS accesses the stored device configurations and subsequently programs each device accordingly. The system BIOS does not perform any conflict detection or resolution.

### Dynamic Resource Allocation

The method for dynamic resource allocation is for the system BIOS POST to dynamically allocate resources to configurable devices using a procedure considered most desirable or effective to the system BIOS. This method usually needs to know what resources are being used by static (old ISA) devices in the system to work successfully. The system resources allocated to the static devices are registered with the system BIOS through function 09h, ***Set Statically Allocated Resource Information***, of the runtime services.
The primary benefits of dynamic resource allocation are the minimal amount of nonvolatile storage required and the flexibility in resource allocation provided to the Plug and Play devices installed in the system. As mentioned above, the system BIOS needs to know the system resources being used by the static devices for effective dynamic resource allocation. The system software must provide this information through the ***Set Statically Allocated Resource Information*** function.
*Example:* An example of a system which supports dynamic resource allocation is one where the system BIOS only stores information regarding static ISA devices (assuming that this information is supplied by a configuration utility). Using this stored information, a system BIOS could use semi-intelligent algorithms to configure the Plug and Play devices "around" the static ISA devices. Such a configuration is dynamic because it is determined each time the system boots.

## 2.2 Plug and Play ISA Card Support

One responsibility of a Plug and Play BIOS during POST is to isolate and initialize all Plug and Play ISA cards and assign them with a valid Card Select Number (***CSN)***. Once a ***CSN*** is assigned, the system BIOS can then designate resources to the Plug and Play ISA cards according to the resource allocation scheme chosen for the system. While the configuration of the required Plug and Play ISA boot devices by the Plug and Play BIOS is mandatory, all of the remaining Plug and Play devices may be configured dynamically by the system software at boot. The system BIOS mayalso provide a mechanism for system software to explicitly allocate system resources to the Plug and Play ISA cards in the system. For example, the system BIOS could provide support for allocating the last working configuration.

### 2.2.1 Assigning CSN to Plug and Play ISA cards

Early in the POST process, a Plug and Play system BIOS should always perform the isolation process for Plug and Play ISA cards as specified in the Plug and Play ISA specification V1.00. This process should be performed regardless if ***CSNs*** have already been assigned to the Plug and Play Devices. This will guarantee accurate initialization of each Plug and Play device during the start of the operating system.
The Plug and Play ISA specification requires that ***CSNs*** must be assigned sequentially starting at one and continuing in the order that each Plug and Play ISA card is isolated.
A responsibility of the system BIOS is to maintain the last assigned ***CSN***. This information will be returned through function 40h, ***Get ISA Configuration Structure,*** of the Plug and Play runtime services. Programs that want to scan through the ***CSNs*** looking for their adapter will need to know the last ***CSN*** assigned.

On systems with a dynamic ISA bus, like portables, function 40h will be more flexible. When an ISA bus is present, the information returned by function 40h will always be valid after a cold boot. On a cold boot with no ISA bus present, function 40h will return zeros. After an ISA warm/hot dock, the function 40h information will also be valid, if the plug and play BIOS isolates and enumerates the plug and play adapter cards before returning control to the plug and play operating system. If the BIOS does not re-enumerate after an ISA warm/hot dock event, then the information returned by function 40h will be zeros. After an ISA undock event, this information will also be zeros.

### 2.2.2 Initializing Plug and Play ISA Cards

After CSNs have been assigned, all Plug and Play ISA devices should be inactive. Later in POST when the system resources have been determined, Plug and Play ISA cards will be enabled as determined by the system's allocation scheme. This means that at least all of the Plug and Play ISA bootable cards will be configured and enabled.

During the POST sequence, the system BIOS will need to select an Input, Output, and Initial Program Load (IPL) device. Based on the other devices in the system, any Plug and Play device that is a boot device will get enabled to provide the boot services. Plug and Play devices that are not boot devices may get enabled later in POST if, and only if they can be enabled without creating a resource conflict.

The method used to allocate resources to the Plug and Play ISA cards depends on the resource allocation method described in the section above. If ***Static Resource Allocation*** is being used then the Plug and Play ISA devices will be initialized according to the information specified for the last working configuration. If ***Dynamic Resource Allocation*** is being used then resource information available from the Plug and Play ISA card will be used to configure the device during the BIOS POST process.

## 2.3 BIOS POST Option ROM Initialization

One of the new features of the Plug and Play BIOS architecture is the enhancements to the ISA Option ROM architecture. This new interface will help couple the system BIOS closely with the Plug and Play option ROM to assist the system BIOS in completing the POST configuration process. For details about the Plug and Play option ROM enhancements, refer to the section on the Plug and Play Option ROM. This section describes how the system BIOS will initialize both standard ISA and Plug and Play Option ROMs.

All ISA option ROMs that are not Plug and Play compatible will be initialized by the Plug and Play BIOS POST using the exact procedure used in existing PC compatible systems. This procedure is performed by scanning the C0000h to EFFFFh address space on 2K boundaries searching for a 55AAh header. Once located, the module is checksummed to determine if the structure is valid and, if valid, the option ROM is initialized by making a far call to offset 03h within the segment.

There are two different environments that Plug and Play compliant option ROMs could be installed in. The first is a standard PC compatible system that does not have a Plug and Play compatible system BIOS. The second environment is a system that has a Plug and Play system BIOS. The option ROM can determine which environment it is installed in by examining the register information passed to the option ROM's initialization routine. It is able to perform this check because the Plug and Play BIOS will provide the following information:

Entry: ES:DI Pointer to System BIOS Plug and Play Installation Check Structure (See Section 4.4)
The following registers will only be initialized for Plug and Play ISA devices:
BX Card Select Number for this card, FFFFh if this device is not ISA Plug and Play.
DX Read Data Port address, FFFFh if there are no ISA Plug and Play devices in the system.

For other bus architectures, refer to the appropriate specification. For example, the PCI Local Bus Specification R2.0 published by the PCI SIG specifies AH=Bus number and AL=Device Function number as parameters for Option ROM initialization.

On a system that does not have a Plug and Play compatible system BIOS, ES:DI would not point to a valid Plug and Play Installation Check Structure . Therefore, by validating the contents of the data pointed to in ES:DI, the option ROM can determine whether it is being initialized from a Plug and Play or non-Plug

and Play system BIOS. Once the option ROM has determined the environment it is installed in, it can perform the proper steps for initialization.
In the first environment, which is a standard PC compatible system that does not have a Plug and Play compatible system BIOS, the ISA option ROM scan will be performed and the Plug and Play option ROM should initialize exactly as if it was a standard ISA option ROM.
In the second environment, where the system has a Plug and Play system BIOS, the option ROM will recognize the Plug and Play installation check structure and perform the initialization as specified in section 3, which describes the option ROM support. Option ROM initialization routines can not depend on any of the Plug and Play runtime functions to be available until after INT19 has been invoked at the end of the POST process.2.4 Interrupt 19H Execution
Interrupt 19h, commonly referred to as the system bootstrap loader, is responsible for loading and executing the first sector of the operating system. This bootstrap sequence is the final component of the system BIOS POST before control is passed onto the operating system. In a PC system, the Initial Program Load (IPL) device can easily be any device supported by an option ROM if it intercepts Interrupt 13h and provides these services. However, some option ROMs have gone even further and captured Interrupt 19h to control the bootstrap process.
An Option ROM which takes control of Interrupt 19h presents a major problem to a Plug and Play system BIOS. The system BIOS can no longer control which device will be the Initial Program Load (IPL) device since it no longer controls the bootstrap sequence. Given this dilemma, the system BIOS POST will recapture Interrupt 19h away from an option ROM if the primary Initial Program Load (IPL) device is either a Plug and Play ISA device or a device that is known to the system BIOS (e.g., ATA compatible IDE fixed disk).
One particularly interesting situation occurs when the system BIOS has recaptured Interrupt 19h and then determines that it cannot load the operating system due to invalid media or other problems. In this case, the Plug and Play system BIOS will restore the last captured Interrupt 19h vector and reinitiate the Interrupt 19h boot sequence.

## 2.4 Transferring Control to the Operating System

The very last function of the system BIOS POST after loading and validating the operating system boot sector is to transfer control. In an ISA system, control is transferred without any parameters. In a Plug and Play system BIOS, parameters will be passed to the operating system. The parameters are:

Entry: ES:DI Pointer to System BIOS Plug and Play Installation Check Structure (See section 4.4)
DL Physical device number the OS is being loaded from (e.g. 80h, assuming the device supports INT 13H interface.)

In a non-Plug and Play operating environment this information will have no meaning. However, a Plug and Play operating system will look for a Plug and Play system BIOS and use any information it may need. The physical device number is passed to allow the operating system to continue to load from the current physical device, instead of assuming a physical device of 00h or 80h.

## 2.5 POST Execution flow

The following steps outline a typical flow of a Plug and Play system BIOS POST. All of the standard ISA functionality has been eliminated for clarity in understanding the Plug and Play POST enhancements.

***Step 1*** **Disable all configurable devices**
Any configurable devices known to the system BIOS should be disabled early in the POST process.

***Step 2*** **Identify all Plug and Play ISA devices**
Assign CSNs to Plug and Play ISA devices but keep devices disabled. Also determine which devices are boot devices.

***Step 3*** **Construct an initial resource map of allocated resources**
Construct a resource map of resources that are statically allocated to devices in the system. If the system software has explicitly specified the system resources assigned to ISA devices in the

system through the ***Set Statically Allocated Resource Information*** function, the system BIOS will create an initial resource map based on this information.
If the BIOS implementation provides support for saving the last working configuration and the system software has explicitly assigned system resources to specific devices in the system, then this information will be used to construct the resource map. This information will also be used to configure the devices in the system.

***Step 4*** **Enable Input and Output Devices**
Select and enable the Input and Output Device. Compatibility devices in the system that are not configurable always have precedence. For example, a standard VGA adapter would become the primary output device. If configurable Input and Output Devices exists, then enable these devices at this time. If Plug and Play Input and Output Devices are being selected, then initialize the option ROM, if it exists, using the Plug and Play option ROM initialization procedure (See section 3).

***Step 5*** **Perform ISA ROM scan**
The ISA ROM scan should be performed from C0000h to EFFFFh on every 2K boundary. Plug and Play Option ROMs are disabled at this time (except input and output boot devices) and will not be included in the ROM scan.

***Step 6*** **Configure the IPL device**
If a Plug and Play device has been selected as the IPL device, then use the Plug and Play Option ROM procedure to initialize the device. If the IPL device is known to the system BIOS, then ensure that interrupt 19h is still controlled by the system BIOS. If not, recapture interrupt 19h and save the vector.

***Step 7*** **Enable Plug and Play ISA and other Configurable Devices**
If a static resource allocation method is used, then enable the Plug and Play ISA cards with conflict free resource assignments. Initialize the option ROMs and pass along the defined parameters. All other configurable devices should be enabled, if possible, at this time.
If a dynamic resource allocation method is used, then enable the bootable Plug and Play ISA cards with conflict free resource assignments and initialize the option ROMs.

***Step 8*** **Initiate the Interrupt 19H IPL sequence**
Start the bootstrap loader. If the operating system fails to load and a previous ISA option ROM had control of the interrupt 19h vector, then restore the interrupt 19h vector to the ISA option ROM and re-execute the Interrupt 19h bootstrap loader.

***Step 9*** **Operating system takes over resource management**
If the loaded operating system is Plug and Play compliant, then it will take over management of the ***system resources.*** It will use the runtime services of the system BIOS to determine the current allocation of these resources. It is assumed that any unconfigured Plug and Play devices will be configured by the appropriate system software or the Plug and Play operating system.

# 3.0 Option ROM Support

This section outlines the Plug and Play Option ROM requirements. This Option ROM support is directed specifically towards boot devices; however, the ***Static Resource Information Vector*** permits non-Plug and Play devices which have option ROMs to take advantage of the Plug and Play Option ROM expansion header to assist a Plug and Play environment whether or not it is a boot device. A boot device is defined as any device which must be initialized prior to loading the Operating System. Strictly speaking, the only required boot device is the Initial Program Load (IPL) device upon which the operating system is stored. However, the definition of boot devices is extended to include a primary Input Device and a primary Output device. In some situations these I/O devices may be required for communication with the user. All new Plug and Play devices that support Option ROMs should support the Plug and Play Option ROM Header. In addition, all non-Plug and Play devices may be "upgraded" to support the Plug and Play Option ROM header as well. While these static ISA devices will still not have software configurable resources, the Plug and Play Option ROM Header will greatly assist a Plug and Play System BIOS in identification and selection of the primary boot devices.
It is important to note that the Option ROM support outlined here is defined specifically for computing platforms based on the Intel X86 family of microprocessors and may not apply to systems based on other types of microprocessors.

# 3.1 Option ROM Header

The Plug and Play Option ROM Header follows the format of the Generic Option ROM Header extensions described in Appendix A. The Generic Option ROM header is a mechanism whereby the standard ISA Option ROM header may be expanded with minimal impact upon existing Option ROMs. The pointer at offset 1Ah may point to ANY type of header. Each header provides a link to the next header; thus, future Option ROM headers may use this same generic pointer and still coexist with the Plug and Play Option ROM header. Each Option ROM header is identified by a unique string. The length and checksum bytes allow the System BIOS and/or System Software to verify that the header is valid.
Standard Option ROM Header:

| Offset | Length | Value | Description | |
|---|---|---|---|---|
| 0h | 2h | AA55h | Signature | Standard |
| 2h | 1h | Varies | Option ROM Length | Standard |
| 3h | 4h | Varies | Initialization Vector | Standard |
| 7h | 13h | Varies | Reserved | Standard |
| 1Ah | 2h | Varies | Offset to Expansion Header Structure | New for Plug and Play |

**Signature** - All ISA expansion ROMs are currently required to identify themselves with a signature WORD of AA55h at offset 0. This signature is used by the System BIOS as well as other software to identify that an Option ROM is present at a given address.
**Length** - The length of the option ROM in 512 byte increments.
**Initialization vector** - The system BIOS will execute a FAR CALL to this location to initialize the Option ROM. A Plug and Play System BIOS will identify itself to a Plug and Play Option ROM by passing a pointer to a Plug and Play Identification structure when it calls the Option ROM's initialization vector. If the Option ROM determines that the System BIOS is a Plug and Play BIOS, the Option ROM should not hook the input, display, or IPL device vectors (INT 9h, 10h, or 13h) at this time. Instead, the device should wait until the System BIOS calls the Boot Connection vector before it hooks any of these vectors. *Note: A Plug and Play device should never hook INT 19h or INT 18h until its Boot Connection Vector, offset 16h of the Expansion Header Structure (section 3.2), has been called by the Plug and Play system BIOS.*
If the Option ROM determines that it is executing under a Plug and Play system BIOS, it should return some device status parameters upon return from the initialization call. See the section on Option ROM Initialization for further details.
The field is four bytes wide even though most implementations may adhere to the custom of defining a simple three byte NEAR JMP. The definition of the fourth byte may be OEM specific.
**Reserved** - This area is used by various vendors and contains OEM specific data and copyright strings.
**Offset to Expansion Header** - This location contains a pointer to a linked list of Option ROM expansion headers. Various Expansion Headers (regardless of their type) may be chained together and accessible via this pointer. The offset specified in this field is the offset from the start of the option ROM header.

## 3.2 Expansion Header for Plug and Play

| Offset | Length | Value | Description | |
|---|---|---|---|---|
| 0h | 4 BYTES | $PnP (ASCII) | Signature | Generic |
| 04h | BYTE | Varies | Structure Revision | 01h |
| 05h | BYTE | Varies | Length (in 16 byte increments) | Generic |
| 06h | WORD | Varies | Offset of next Header (0000h if none) | Generic |
| 08h | BYTE | 00h | Reserved | Generic |
| 09h | BYTE | Varies | Checksum | Generic |
| 0Ah | DWORD | Varies | Device Identifier | PnP Specific |
| 0Eh | WORD | Varies | Pointer to Manufacturer String (Optional) | PnP Specific |
| 10h | WORD | Varies | Pointer to Product Name String (Optional) | PnP Specific |
| 12h | 3 BYTE | Varies | Device Type Code | PnP Specific |
| 15h | BYTE | Varies | Device Indicators | PnP Specific |
| 16h | WORD | Varies | Boot Connection Vector - Real/Protected mode (0000h if none) | PnP Specific |
| 18h | WORD | Varies | Disconnect Vector - Real/Protected mode (0000h if none) | PnP Specific |
| 1Ah | WORD | Varies | Bootstrap Entry Point - Real/Protected mode (0000h if none) | PnP Specific |
| 1Ch | WORD | 0000h | Reserved | PnP Specific |
| 1Eh | WORD | Varies | Static Resource Information Vector- Real/Protected mode (0000h if none) | PnP Specific |

**Signature** - All Expansion Headers will contain a unique expansion header identifier. The Plug and Play expansion header's identifier is the ASCII string "$PnP" or hex 24 50 6E 50h (Byte 0 = 24h ... Byte 3 = 50h).
**Structure Revision** - This is an ordinal value that indicates the revision number of this structure only and does not imply a level of compliance with the Plug and Play BIOS version.
**Length** - Length of the entire Expansion Header expressed in sixteen byte blocks. The length count starts at the Signature field.
**Offset of Next Header** - This location contains a link to the next expansion ROM header in this Option ROM. If there are no other expansion ROM headers, then this field will have a value of 0h. The offset specified in this field is the offset from the start of the option ROM header.
**Reserved** - Reserved for Expansion
**Checksum** - Each Expansion Header is checksummed individually. This allows the software which wishes to make use of an expansion header (in this case, the system BIOS) the ability to determine if the expansion header is valid. The method for validating the checksum is to add up all byte values in the Expansion Header, including the *Checksum* field, into an 8-bit value. A resulting sum of zero indicates a valid checksum operation.
**Device Identifier** - This field contains the Plug and Play Device ID.
**Pointer to Manufacturer String (Optional) -** This location contains an offset relative to the base of the Option ROM which points to an ASCIIZ representation of the board manufacturer's name. This field is optional and if the pointer is 0 (Null) then the Manufacturer String is not supported.
**Pointer to Product Name String (Optional) -** This location contains an offset relative to the base of the Option ROM which points to an ASCIIZ representation of the product name. This field is optional and if the pointer is 0 (Null) then the Product Name String is not supported.
**Device Type Code** - This field contains general device type information that will assist the System BIOS in prioritizing the boot devices.
The Device Type code is broken down into three byte fields. The byte fields consist of a Base-Type code that indicates the general device type. The second byte is the device Sub-Type and its definition is

dependent upon the Base-Type code. The third byte defines the specific device programming interface, IF.-Type, based on the Base-Type and Sub-Type.
Refer to Appendix B for a description of Device Type Codes.
**Device Indicators** - This field contains indicator bits that identify the device as being capable of being one of the three identified boot devices: Input, Output, or Initial Program Load (IPL).

| Bit | Description |
|---|---|
| 7 | A 1 indicates that this ROM supports the Device Driver Initialization Model |
| 6 | A 1 indicates that this ROM may be Shadowed in RAM |
| 5 | A 1 indicates that this ROM is Read Cacheable |
| 4 | A 1 indicates that this option ROM is only required if this device is selected as a boot device. |
| 3 | Reserved (0) |
| 2 | A 1 in this position indicates that this device is an Initial Program Load (IPL) device. |
| 1 | A 1 in this position indicates that this device is an Input device. |
| 0 | A 1 in this position indicates that this device is a Display device. |

**Boot Connection Vector** (Real/Protected mode) - This location contains an offset from the start of the option ROM header to a routine that will cause the Option ROM to hook one or more of the primary input, primary display, or Initial Program Load (IPL) device vectors (INT 9h, INT 10h, or INT 13h), depending upon the parameters passed during the call.
When the system BIOS has determined that the device controlled by this Option ROM will be one of the boot devices (the Primary Input, Primary Display, or IPL device), the System ROM will execute a FAR CALL to the location pointed to by the Boot Connection Vector. The system ROM will pass the following parameters to the options ROM's Boot Connection Vector:

| Reg On Entry | Description |
|---|---|
| AX | Provides an indication as to which vectors should be hooked by specifying the type of boot device this device has been selected as.<br>Bit 7..3 Reserved(0)<br>Bit 2 1=Connect as IPL (INT 13h)<br>Bit 1 1=Connect as primary Video (INT 10h)<br>Bit 0 1=Connect as primary Input (INT 09h) |
| ES:DI | Pointer to System BIOS PnP Installation Check Structure (See section 4.4) |
| BX | CSN for this card, ISA PnP devices only. If not an ISA PnP device then this parameter will be set to FFFFh. |
| DX | Read Data Port, (ISA PnP devices only. If no ISA PnP devices then this parameter will be set to FFFFh. |

**Disconnect Vector** (Real/Protected mode) - This vector is used to perform a cleanup from an unsuccessful boot attempt on an IPL device. The system ROM will execute a FAR CALL to this location on IPL failure.

**Bootstrap Entry Vector** (Real/Protected mode) - This vector is used primarily for RPL (Remote Program Load) support. To RPL (bootstrap), the System ROM will execute a FAR CALL to this location. The System ROM will call the Real/Protected Mode Bootstrap Entry Vector instead of INT 19h if:

a) The device indicates that it may function as an IPL device,
b) The device indicates that it does not support the INT 13h Block Mode interface,
c) The device has a non-null Bootstrap Entry Vector,
d) The Real/Protected Mode Boot Connection Vector is null.

The method for supporting RPL is beyond the scope of this specification. A separate specification should define the explicit requirements for supporting RPL devices.

**Reserved** - Reserved for Expansion

**Static Resource Information Vector** - This vector may be used by non-Plug and Play devices to report static resource configuration information. Plug and Play devices should not support the Static Resource Information Vector for reporting their configuration information. This vector should be callable both before and/or after the option ROM has been initialized. The call interface for the Static Resource Information Vector is as follows:

Entry: ES:DI Pointer to memory buffer to hold the device's static resource configuration information. The buffer should be a minimum of 1024 bytes. This information should follow the System Device Node data structure, except that the *Device node number* field should always be set to 0, and the information returned should only specify the currently allocated resources *(Allocated resource configuration descriptor block)* and not the block of possible resources *(Possible resource configuration descriptor block).* The *Possible resource configuration descriptor block* should only contain the *END_TAG* resource descriptor to indicate that there are no alternative resource configuration settings for this device because the resource configuration for this device is static. Refer to the *Plug and Play ISA Specification* under the section labeled *Plug and Play Resources* for more information about the resource descriptors. This data structure has the following format:

| Field | Size |
|---|---|
| Size of the device node | WORD |
| Device node number/handle | BYTE |
| Device product identifier | DWORD |
| Device type code | 3 BYTES |
| Device node attribute bit-field | WORD |
| Allocated resource configuration descriptor block | VARIABLE |
| Possible resource configuration descriptor block - should only specify the *END_TAG* resource descriptor | 2 BYTES |
| Compatible device identifiers | VARIABLE |

Refer to section 4.2 for a complete description of the elements that make up the System Device Node data structure.
For example, an existing, non-Plug and Play SCSI card vendor could choose to rev the SCSI board's Option ROM to support the Plug and Play Expansion Header. While this card wouldn't gain any of the configuration benefits provided to full hardware Plug and Play cards, it would allow Plug and Play software to determine the devices configuration and thus ensure that Plug and Play cards will map around the static SCSI board's allocated resources.

## 3.3 Option ROM Initialization

The System BIOS will determine if the Option ROM it is about to initialize supports the Plug and Play interface by verifying the Structure Revision number in the device's Plug and Play Header Structure. For all Option ROMs compliant with the 1.0 Plug and Play BIOS Specification, the System BIOS will call the device's initialization vector with the following parameters:

| Reg On Entry | Description |
|---|---|
| ES:DI | Pointer to System BIOS PnP Installation Check Structure (See section 4.4) |
| BX | CSN for this card, ISA PnP devices only. If not an ISA PnP device then this parameter will be set to FFFFh. |
| DX | Read Data Port, (ISA PnP devices only. If no ISA PnP devices then this parameter will be set to FFFFh. |

For other bus architectures refer to the appropriate specification for register parameters on entry.
During initialization, a Plug and Play Option ROM may hook any vectors and update any data structures required for it to access any attached devices and perform the necessary identifications and initializations. However, upon exit from the initialization call, the Option ROM must restore the state of any vectors or data structures related to boot devices (INT 9h, INT 10h, INT 13h, and associated BIOS Data Area [BDA] and Extended BIOS Data Area [EBDA] variables).

Upon exit from the initialization call, Plug and Play Option ROMs should return some boot device status information in the following format:
Return Status from Initialization Call:

| AX Bit | Description |
|---|---|
| 8 | 1 = IPL Device supports INT 13h Block Device format |
| 7 | 1 = Output Device supports INT 10h Character Output |
| 6 | 1 = Input Device supports INT 9h Character Input |
| 5:4 | 00 = No IPL device attached<br>01 = Unknown whether or not an IPL device is attached<br>10 = IPL device attached<br>(RPL devices have a connection).<br>11 = Reserved |
| 3:2 | 00 = No Display device attached<br>01 = Unknown whether or not a Display device is attached<br>10 = Display device attached<br>11 = Reserved |
| 1:0 | 00 = No Input device attached<br>01 = Unknown whether or not an Input device is attached<br>10 = Input device attached<br>11 = Reserved |

## 3.4 Option ROM Initialization flow

The following outlines the typical steps used to initialize Option ROMs during a Plug and Play system BIOS POST:

***Step 1*** **Initialize the boot device option ROMs.**
This includes the Primary Input, Primary Output, and Initial Program Load (IPL) device option ROMs.

***Step 2*** **Initialize ISA option ROMs by performing ISA ROM scan**
The ISA ROM scan should be performed from C0000h to EFFFFh on every 2k boundary. Plug and Play option ROMs will not be included in the ROM scan.

***Step 3*** **Initialize option ROMs for ISA devices which have a Plug and Play option ROM.**
Typically, these devices will not provide support for dynamic configurability. However, the resources utilized by these devices can be obtained through the *Static Resource Information Vector* as described in section 3.2.

***Step 4*** **Initialize option ROMs for Plug and Play cards which have a Plug and Play option ROM.**

***Step 5*** **Initialize option ROMs which support the Device Driver Initialization Model (DDIM).**
Option ROMs which follow this model make the most efficient use of space consumed by option ROMs. Refer to Appendix B for more information on the DDIM.

## 3.5 ISA Option ROMs and Resource Mapping

Given the fact that add-in cards are encouraged to make all of their resource assignments flexible, there arises an interesting issue for Option ROMs, in how does the Option ROM code "know" which resource values to use to communicate with the card? There are several possible solutions to this problem, but the one selected for Plug and Play Option ROMs is as follows.
When the Plug and Play Option ROM is initialized, it will be passed the CSN and Read Data Port. The Option ROM can use this information to determine which resources were assigned to it. When the Option ROM has determined this, it should then setup its entry vectors based upon the resource assignment. For example, if an add-in SCSI controller has two possible I/O Port assignments, 300h and 310h, then it should have two different entry vectors for INT 13h. Depending upon which base I/O address is assigned, the Option ROM will setup the INT 13h vector to point to the proper entry vector. Thereafter, whenever

INT 13h is called, the Option ROM may make the assumption that the base I/O address is the one that goes with that entry point.

## 3.6 Error Recovery: Returning to the Boot flow

In the current boot model for standard PC compatible systems, once the system BIOS turns control over to the Initial Program Load (IPL) device's boot sector, there is no way for the boot sector to return control to the system BIOS in the event that an OS loader is not present on the disk, or the IPL fails for some other reason.

In the Plug and Play Boot model, an attempt is made to correct this. If at any time after control has been turned over to the IPL device's boot sector either the boot sector or some other portion of the OS loader determines that the IPL device is incapable of supporting the boot process, control may be returned to the system BIOS (so that the system BIOS can attempt to boot off of a different IPL device) by issuing either an INT 19h or an INT 18h. The BIOS will intercept this INT vector and attempt to continue the bootstrap process.

# 4.0 Configuration Support

A Plug and Play system BIOS, in addition to providing a conflict free bootstrap process, also provides services to the operating system to assist with resource management during runtime. These services focus on extending Plug and Play support to non-Plug and Play systemboard devices and dynamic event notification.

## 4.1 System Device Configuration List

The system device configuration list consists of nodes or data structures that identify the embedded devices that are on the system. The embedded devices consist of systemboard components that provide the base functionality for the system. This includes devices such as the Programmable Interrupt Controller (PIC), the DMA Controller, System Timer, Keyboard Controller, Integrated Video Controller, Floppy Controller, etc. The system device configuration list only provides information about the systemboard devices and does not include nodes for devices plugged into an expansion bus. The system device configuration list does not identify the peripherals that are attached to the embedded systemboard devices. For instance, the system configuration list will identify an integrated fixed disk controller but will not provide nodes for any fixed disk drives that might be attached to the controller. It is assumed that peripherals will be identified by other software. The system BIOS provides an interface for system software to access the information in the system configuration list through the BIOS functions that are defined later in this document. The System Device Node data structure provides configuration information about a single systemboard device. The information returned for each systemboard component reported through the Plug and Play BIOS interface will follow the data structure format specified for the System Device Node. The next subsection describes the System Device Node data structure.

## 4.2 System Device Node Definition

The System Device Node is the structure that represents a single embedded systemboard device. The elements that make up this structure provide information that describe the device and the system resources that have been allocated to the device. This includes reporting the system resources that have typically been reserved for standard PC compatible systemboard devices, such as I/O port addresses from 00h to FFh. The information for alternative or possible resource configuration settings can be provided in the System Device Node; however, it is not required. The various possible resource settings can also be provided in a configuration file or an image of the configuration file, in ROM, supplied by the system vendor. This configuration file would contain the necessary configuration information not contained in the System Device Node, and can provide more information to the user about the specific devices. If the configuration information is contained in both the System Device Node and in a configuration file, then the system resources possibilities for the device that are specified in the configuration file should take precedence over the information contained in the system device node. The following data structure defines the required elements for the base System Device Node. Please refer to the *Plug and Play ISA Specification version 1.0A* (Section 4.6) for the maximum resources that a device node can use.

| Field | Size |
|---|---|
| Size of the device node | WORD |
| Device node number/handle | BYTE |
| Device product identifier | DWORD |
| Device type code | 3 BYTES |
| Device node attribute bit-field | WORD |
| Allocated resource configuration descriptor block | VARIABLE |
| Possible resource configuration descriptor block | VARIABLE |
| Compatible device identifiers | VARIABLE |

***Size of Device Node:***
This field contains the size, in bytes, of the device node.
***Device node number:***
The node number, or handle, is a unique identifier value assigned to the node by the system BIOS and is used to access the node information through the BIOS interface.
***Device product identifier:***
This field is an EISA ID, which is a seven character ASCII representation of the product identifier compressed into a 32-bit identifier. The seven character ID consists of a three character manufacturer code, a three character hexadecimal product identifier, and a one character hexadecimal revision number. For example, the third revision of the ABC device might have an uncompressed ID such as ABC1003.
The manufacturer code is a 3 uppercase character code that is compressed into 3 5-bit values as follows:
1. Find hex ASCII value for each letter
2. Subtract 40h from each ASCII value
3. Retain 5 least-significant bits for each letter by discarding upper 3-bits because they are always 0.
4. Compressed code = Concatenate 0 and the 3 5-bit values for the character.

The format of the compressed product identifier is as follows:

| Byte | Description |
|---|---|
| 0 | Bit 7: Reserved (0)<br>Bits 6-2: 1st character of the compressed manufacturer code<br>Bits 1-0: Upper 2 bits of the 2nd character of the compressed manufacturer code |
| 1 | Bits 7-5: Lower 3 bits of the 2nd character of the compressed manufacturer code.<br>Bits 4-0: 3rd character of the compressed manufacturer code.<br>(bit 4 is most significant) |
| 2 | Bits 7-4: 1st hexadecimal digit of the product number. (bit 7 is most significant)<br>Bits 3-0: 2nd hexadecimal digit of the product number. (bit 3 is most significant) |
| 3 | Bits 7-4: 3rd hexadecimal digit of the product number (bit 7 is most significant)<br>Bits 3-0: Hexadecimal digit for the revision number. (bit 3 is most significant) |

Refer to the *Device Identifier Reference Table & Device Type Code Table* for a list of product identifiers. This list includes generic Plug and Play device identifiers for the standard systemboard components. See the References section of this document.
***Device type code:***
This field is used to specify the type or characteristics of the node in the configuration list. There are many different kinds of controllers and devices and through the type field you can identify which kind of component this node represents.
The Device Type code is broken down into three byte fields. The first byte in the Device Type Code consists of a Base Type code which indicates the general device type. The second byte is the device Sub-Type and its definition is dependent upon the Base Type code. The third byte defines the specific device programming interface, IF. Type, based on the Base Class and Sub-Class.
Refer to the *Device Identifier Reference Table & Device Type Code Table* for a description of Device Type Codes.
***Device node attribute bit-field:***

The device node attributes provide additional information about the state of the device and the capabilities of the device. This bit-field is defined as follows:

bit 15-9: reserved (0)
bits 8:7 0:0=device can only be configured for next boot (static)
0:1=device can be configured at runtime (dynamically)
1:0=Reserved
1:1=device can only be configured at runtime (dynamically)
bit 6: 0=device is not a removable system device
1=device is a removable system device
bit 5: 0=device is not a docking station device
1=device is a docking station device
bit 4: 0=device is not capable of being primary Initial Program Load (IPL) device
1=device is capable of being primary IPL device
bit 3: 0=device is not capable of being primary input device
1=device is capable of being primary input device
bit 2: 0=device is not capable of being primary output device
1=device is capable of being primary output device
bit 1: 0=device is configurable
1=device is not configurable
bit 0: 0=device can be disabled
1=device cannot be disabled

Bit 0 specifies whether the device can be disabled or not. If the device is disabled, it is assumed that the system resources that the device was using are available for use by other devices.
Bit 1 indicates that the device is configurable. This implies that the system device node provides the resource requirements for the device in the *Possible resource configuration descriptor block*. If the device node does not specify the resource requirements or the device does not have any alternate system resource requirements, bit 1 must be set to indicate that the device is not configurable.
Bits 2-4 identify the capability of the device being designated as a boot device.
Bit 5 indicates that the device resides on a docking station or convenience base.
Bit 6 indicates that the device node represents a device that is removable on the base system unit, such as a removable floppy drive.

Bits 8:7 use three of the four possible states to indicate if the device node can be configured dynamically, configured statically only for next boot or configured dynamically only. The fourth state is reserved.

***Allocated resource configuration descriptor block:***

The allocated resource descriptor block describes the system resources allocated to this device. The format of the data contained in this block follows the format defined in the *Plug and Play ISA Specification* under the section labeled *Plug and Play Resources*. The resource data is provided as a series of data structures with each of the resource data structures having a unique tag or identifier. These are the resource descriptors which specifically describe the standard PC system resources, such as Memory, I/O addresses, IRQs, and DMA channels.

***Possible resource configuration descriptor block:***

The alternative resource selections that a particular device can support can be obtained from the data contained in this block. The format of the data in this block follows the same format as the allocated resource descriptor block. Refer to the *Plug and Play ISA Specification* under the section labeled *Plug and Play Resources* for a description of the data structures that make up the resource descriptor blocks. These are the resource descriptors which specifically describe the standard PC system resources, such as Memory, I/O addresses, IRQs, and DMA channels.
The information in this block can be used by the system BIOS and/or system software for selecting a conflict free resource allocation for this device without user intervention. The data in this block is optional. If this information is not provided in this structure, it can optionally be provided in a configuration file for the systemboard that defines the configuration information for the embedded devices. If the possible resource configurations are not specified in either place the device is assumed to

be a static device, which means it is not configurable. If the information is provided in this descriptor block and in a configuration file, the possible resource selections must be specified in the same order that they are described in the configuration file. If the node does not contain the alternative resource selections then the first byte in this block will contain the *End Tag descriptor*, which is described in *Plug and Play ISA Specification*, to indicate that there are no resources in this block.

***Compatible device identifiers:***

The compatible device identifiers block specifies the IDs of other devices that this device is compatible with. System software can use this information to load compatible device drivers if necessary. The format of the data contained in this block follows the format defined in the *Plug and Play ISA Specification* under the section labeled *Plug and Play Resources - Compatible Device ID*.

## 4.3 Plug and Play BIOS Functions

The following subsections describe the Plug and Play BIOS interface. The function return values are listed in Appendix C. **The Plug and Play BIOS functions will preserve all FLAGS and registers except for the AX register, which will contain the return code.** The BIOS functions will use the caller's stack and a minimum of 1024 bytes of stack space must be available to these functions. **It is important to note that system BIOS function(s) used to set the configuration of a systemboard device will not validate the configuration information passed by the caller and may not return an error code.**

Option ROM initialization routines can not depend on any of the Plug and Play runtime functions to be available until after INT19 has been invoked at the end of the POST process.

## 4.4 Plug and Play Installation Check

This section describes the method for system software to determine if the system has a Plug and Play BIOS. This Plug and Play installation check indicates whether the system BIOS support for accessing the configuration information about the devices on the systemboard is present and the entry point to these BIOS functions. This method involves searching for a signature of the ASCII string $PnP in system memory starting from F0000h to FFFFFh at every 16 byte boundary. This signature indicates the system may have aPlug and Play BIOS and identifies the start of a structure that specifies the entry point of the BIOS code which implements the support described in this document. The system software can determine if the structure is valid by performing a **Checksum** operation.

The method for calculating the checksum is to add up *Length* bytes from the top of the structure, including the *Checksum* field, into an 8-bit value. A resulting sum of zero indicates a valid checksum operation. The entry points specified in this structure are the software interface to the BIOS functions. The structure element that specifies the 16-bit protected mode entry point will allow the caller to construct a protected mode selector for calling this support. The structure of the Plug and Play BIOS Support Installation Check is as follows:

| Field | Offset | Length | Value |
|---|---|---|---|
| Signature | 00h | 4 BYTES | $PnP (ASCII) |
| Version | 04h | BYTE | 10h |
| Length | 05h | BYTE | 21h |
| Control field | 06h | WORD | Varies |
| Checksum | 08h | BYTE | Varies |
| Event notification flag address | 09h | DWORD | Varies |
| Real Mode 16-bit offset to entry point | 0Dh | WORD | Varies |
| Real Mode 16-bit code segment address | 0Fh | WORD | Varies |
| 16-Bit Protected Mode offset to entry point | 11h | WORD | Varies |
| 16-Bit Protected Mode code segment base address | 13h | DWORD | Varies |

| | | | |
|---|---|---|---|
| OEM Device Identifier | 17h | DWORD | Varies |
| Real Mode 16-bit data segment address | 1Bh | WORD | Varies |
| 16-Bit Protected Mode data segment base address | 1Dh | DWORD | Varies |

**Signature** is represented as the ASCII string "$PnP", where byte 0='$' (24h), byte 1='P' (50h),byte 2='n' (6Eh), and byte 3='P' (50h).
**Version** - This is a BCD value that implies a level of compliance with major (high nibble) and minor (low nibble) version changes of the Plug and Play BIOS specification. For example, the BCD value 10h would be interpreted as version 1.0.
**Length** - Length of the entire Installation Structure expressed in bytes. The length count starts at the Signature field.
The **Control field** is a bit-field that provides system capabilities information.

bits 15:2: Reserved (0)
bits 1:0: Event notification mechanism
00=Event notification is not supported
01=Event notification is handled through polling
10=Event notification is asynchronous (at interrupt time)

**Checksum** - The method for calculating the checksum is to add up the number of bytes in the Installation Structure, including the *Checksum* field, into an 8-bit value. A resulting sum of zero indicates a valid checksum.
The **Event notification flag address** specifies the physical address of the *Event Flag* if event notification is handled through polling. When event notification is handled through polling, bit 0 of the *Event Flag* will be set when a system event occurs. System software will monitor or poll the *Event Flag* for notification of an event.
If events are handled through asynchronous notification, the system BIOS will specify a system device node which can be obtained from the *Get Node* runtime function. The system device node for asynchronous event management will be identified through the device identifier field in the device node data structure and will specify the IRQ number and an I/O port address. This event system device node can be defined in one of two ways. First, the device node can follow the generic implementation in which the device identifier is PNP0C03, and the interrupt number and I/O address assigned are system specific. The only requirement with the generic implementation is that the I/O address bit used for detecting the source of the interrupt and clearing the interrupt line is bit 0. If bit 0 of this I/O address is set to 1, then the interrupt was generated due to a system event. The interrupt service routine should reset the interrupt line by clearing bit 0 at the specified I/O address. All other bits read from the I/O address should not be modified. The second way the event system device node can be defined is implementation specific where the system vendor must supply their own device identifier and whatever resources are required for servicing the event interrupt. This method will require a specific device driver associated with the device node identifier to support the event notification interface.
System software should check the ***Control field*** to determine the event notification method implemented on the system.
Refer to the **Event Notification Interface** section for more information on events.
The **Real Mode 16-Bit interface** is basically the segment:offset of the entry point.
The **16-Bit Protected Mode interface** specifies the code segment base address so that the caller can construct the descriptor from this segment base address before calling this support from protected mode. The offset value is the offset of the entry point. It is assumed that the 16-Bit Protected Mode interface is sufficient for 32-Bit Protected Mode callers.
The caller must also construct data descriptors for the functions that return information in the function arguments that are pointers. The only limitation is that the pointer offset can only point to the first 64K bytes of a segment.
If a call is made to these BIOS functions from 32-bit Protected Mode, the 32-bit stack will be used for passing any stack arguments to the Plug and Play BIOS functions. However, it is important to note that the Plug and Play BIOS functions are not implemented as a full 32-bit protected mode interface and will

access arguments on the stack as a 16-bit stack frame. Therefore, the caller must ensure that the function arguments are pushed onto the stack as 16-bit values and not 32-bit values. The stack parameter passing is illustrated in *Figure 4.4.1* below.

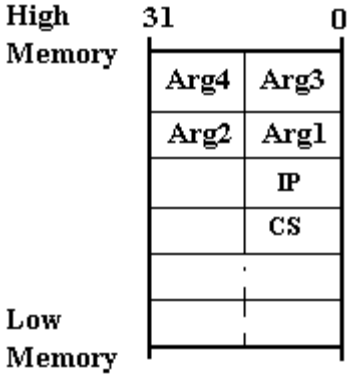


***Figure 4.4.1 - 16-bit Stack Frame on 32-bit Stack***

The Plug and Play system BIOS can determine whether the stack is a 32-bit stack or a 16-bit stack in 16-bit and 32-bit environments through the use of the LAR - Load Access Rights Byte Instruction. The LAR instruction will load the high order doubleword for the specified descriptor. By loading the access rights for the current stack segment selector, the system BIOS can check the B-bit (Big bit) of the stack segment descriptor which identifies the stack segment descriptor as either a 16-bit segment (B-bit clear) or a 32-bit segment (B-bit set).

In addition to executing the LAR command to get the entry point stack size, the BIOS code should avoid ADD BP,X type stack operands in runtime service code paths. These operands carry the risk of faulting if the 32-bit stack base happens to be close to the 64K boundary. For the 16-Bit Protected Mode interface, it is assumed that the segment limit fields will be set to 64K. The code segment must be readable. The current I/O permission bit map must allow accesses to the I/O ports that the system BIOS may need access to in order to perform the function. The current privilege level (CPL) must be less than or equal to I/O privilege level. This will allow the Plug and Play BIOS to use sensitive instructions such as CLI and STI.

The **OEM Device Identifier** field provides a means for specifying a device identifier for the system. The format of the OEM Device Identifier follows the format specified for EISA product identifiers. A system identifier is not required and if not specified, this field should be 0.

The entry point is assumed to have a function prototype of the form,

***int FAR (*entryPoint)(int Function, ...);***

and follow the standard 'C' calling conventions.

System software will interface with all of the functions described in this specification by making a far call to this entry point. As noted above, the caller will pass a function number and a set of arguments based on the function being called. Each function will also include an argument which specifies a data selector which will allow the Plug and Play BIOS to access and update variables within the system BIOS memory space. This data selector parameter is required for protected mode callers. The caller must create a data segment descriptor using the ***16-bit Protected Mode data segment base address*** specified in the Plug and Play Installation Structure, a limit of 64KB, and the descriptor must be read/write capable. Real mode callers are required to set this parameter to the ***Real Mode 16-bit data segment address*** specified in the Plug and Play Installation Structure.

Any functions described by this specification which are not supported should return the FUNCTION_NOT_SUPPORTED return code. The function return codes are described in Appendix C of this specification.

## 4.4.1 System BIOS Plug and Play Compliance - "$PnP"

This section describes the support that is guaranteed by the "$PnP" string in the Plug and Play Installation Check structure and specifies the BIOS support required to be Plug and Play compliant for systems with different characteristics. A Plug and Play compliant system will guarantee:

1. The Plug and Play Structure is valid.

2. Any calls made to the Plug and Play BIOS functions will either perform the function as described by Version 1.0 of this specification or return the FUNCTION_NOT_SUPPORTED error code. Plug and Play compliant systems are required to provide the support as outlined in the table below.

3. All of the runtime Plug and Play services will be contained in a contiguous 64K code segment.

Presence of the $PnP structure in the system BIOS does not mean that the system is fully Plug and Play compliant. For instance, a system BIOS could have a valid $PnP structure; yet, return FUNCTION_NOT_SUPPORTED for each of the functions described in this specification. The following table specifies the required Plug and Play BIOS support necessary for systems with different characteristics to meet full Plug and Play compliance.

| System Characteristics | Required Functions | Optional Functions |
|---|---|---|
| Systems with embedded devices on the systemboard. Proprietary bus devices or local ISA devices on the systemboard. | 00h, 01h, 02h | |
| Systems that support docking to expansion bases | 03h, 04h, 05h | |
| Reserved | | 06h, 07h, 08h |
| Systems with an ISA expansion bus | 40h | 09h, 0Ah |
| ESCD Interface Functions | | 41h, 42h, 43h |
| Systems supporting APM 1.1 (and greater) | 0Bh | |

***Note:***
Functions 09h, 0Ah, and 40h are designed to support systems with an ISA Expansion bus. The information which must be stored in nonvolatile media is the information concerning the resources allocated to static legacy ISA devices. If functions 09h and 0Ah designate that the system implementation utilizes the ESCD for storing static resource allocation, then the caller should utilize the interface defined by the *ESCD Specification* to report statically allocated resources. Functions 41h, 42h, and 43h defined in section 4.7 specify the ESCD interface. Refer to the *ESCD Specification* for a complete description of the interfaces to support the ESCD as well as the format of the ESCD. BIOS support of these functions is optional.Systems with an ISA Expansion bus may provide these BIOS functions to enhance the Plug and Play BIOS POST process for assigning a conflict free configuration to the required boot devices.
The following table provides some examples of systems with certain characteristics and categorically lists the functions that would be required to be Plug and Play compliant.

| Example Systems | Runtime Services | Event | ISA Allocated Resource Support | ISA PnP Isolation |
|---|---|---|---|---|
| Systems without an ISA bus; limited or a variety of boot devices; No Dynamic Events | Required | Not Required | Not Required | Not Required |
| Systems without an ISA bus; limited or a variety of boot devices; Dynamic Events supported | Required | Required | Not Required | Not Required |
| Systems with an ISA bus; No Dynamic Events | Required | Not Required | Not Required | Required |
| Systems with an ISA bus; Dynamic Events supported | Required | Required | Not Required | Required |

## 4.5 System Configuration Interface

The functions described in the following subsections define the System Configuration Interface for obtaining information about the systemboard devices and for setting the system resources utilized by the configurable devices.

### 4.5.1 Function 0 - Get Number of System Device Nodes

#### Synopsis:

```
int FAR (*entryPoint)(Function, NumNodes, NodeSize, BiosSelector);
int Function;                          /* PnP BIOS Function 0 */
unsigned char FAR *NumNodes;           /* Number of nodes the BIOS will return */
unsigned int FAR *NodeSize;            /* Size of the largest device node */
unsigned int BiosSelector;             /* PnP BIOS readable/writable selector */
```

#### Description:

*Required*. This function will return the number of nodes that the system BIOS will return information for in *NumNodes*. These nodes represent only the systemboard devices. In addition to the number of nodes, the system BIOS will return the size, in bytes, of the largest System Device Node in *NodeSize*. This information can be utilized by the system software to determine the amount of memory required to get all of the System Device Nodes.

The *BiosSelector* parameter enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit Protected Mode data segment base address specified in the Plug and Play Installation Check data structure, a limit of 64KB, and the descriptor must be read/write capable. If this function is called from real mode, *BiosSelector* should be set to the Real Mode 16-bit data segment address as specified in the Plug and Play Installation Check structure. Refer to section 4.4 above for more information on the Plug and Play Installation Check Structure and the elements that make up the structure.

The function is available in real mode and 16-bit protected mode.

#### Returns:

0 if successful - SUCCESS

!0 if an error (Bit 7 set) or a warning occurred - error code (The function return codes are described in Appendix C)

The FLAGS and registers will be preserved, except for AX which contains the return code.

#### Example:

The following example illustrates how the 'C' style call interface could be made from an assembly language module:

```
        .
        .
        .
push    Bios Selector
push    segment/selector of NodeSize        ; pointer to NodeSize
push    offset of NodeSize
push    segment/selector of NumNodes        ; pointer to NumNodes
push    offset of NumNodes
push    GET_NUM_NODES                       ; Function 0
call    FAR PTR entryPoint
add     sp,12                               ; Clean up stack
cmp     ax,SUCCESS                          ; Function completed successfully?
jne     error                               ; No-handle error condition
        .
        .
```

### 4.5.2 Function 1 - Get System Device Node

#### Synopsis:

```
int FAR (*entryPoint)(Function, Node, devNodeBuffer, Control, BiosSelector);
int Function;                          /* PnP BIOS Function 1 */
unsigned char FAR *Node;               /* Node number/handle to retrieve */
struct DEV_NODE FAR *devNodeBuffer;    /* Buffer to copy device node data to */
```

*unsigned int Control;* */* Control Flag */*
*unsigned int BiosSelector;* */* PnP BIOS readable/writable selector */*

## Description:

*Required.* This function will copy the information for the specified System Device Node into the buffer specified by the caller. The *Node* argument is a pointer to the unique node number (handle). If *Node* contains 0, the system BIOS will return the first System Device Node. The *devNodeBuffer* argument contains the pointer to the caller's memory buffer. On return, *Node* will be updated with the next node number, or if there are no more nodes, it will contain FFh. The System Device Node data will be placed in the specified memory buffer.

The *Control flag* provides a mechanism for allowing the system software to request a node that indicates either how the specified systemboard device is currently configured or how it is configured for the next boot. *Control* is defined as:

Bits 15:2: Reserved (0)
Bit 1: 0=Do not get the information for how the device will be configured for the next boot.
1=Get the device configuration for the next boot (static configuration).
Bit 0: 0=Do not get the information for how the device is configured right now.
1=Get the information for how the device is configured right now.

If *Control flag* is 0, neither bit 0 nor bit 1 is set, or if both bits are set, this function should return BAD_PARAMETER.

The *BiosSelector* parameter enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit Protected Mode data segment base address specified in the Plug and Play Installation Check data structure, a limit of 64KB, and the descriptor must be read/write capable. If this function is called from real mode, *BiosSelector* should be set to the Real Mode 16-bit data segment address as specified in the Plug and Play Installation Check structure. Refer to section 4.4 above for more information on the Plug and Play Installation Check Structure and the elements that make up the structure.

The function is available in real mode and 16-bit protected mode.

## Returns:

0 if successful - SUCCESS
!0 if an error (Bit 7 set) or a warning occurred - error code (The function return codes are described in Appendix C)

The FLAGS and registers will be preserved, except for AX which contains the return code.

## Example:

The following example illustrates how the 'C' style call interface could be made from an assembly language module:

```
        .
        .
        .
push    Bios Selector
push    Control Flag
push    segment/selector of devNodeBuffer       ; pointer to devNodeBuffer
push    offset of devNodeBuffer
push    segment/selector of Node                ; pointer to Node number
push    offset of Node
push    GET_DEVICE_NODE                         ; Function 1
call    FAR PTR entryPoint
add     sp,14                                   ; Clean up stack
cmp     ax,SUCCESS                              ; Function completed successfully?
jne     error                                   ; No-handle error condition
        .
        .
        .
```

### 4.5.3 Function 2 - Set System Device Node

## Synopsis:

```
int FAR (*entryPoint)(Function, Node, devNodeBuffer, Control, BiosSelector);
int Function;                          /* PnP BIOS Function 2 */
unsigned char  Node;                   /* Node number/handle to set */
struct DEV_NODE FAR *devNodeBuffer;    /* Buffer containing device node data */
unsigned int Control;                  /* Control Flag */
 unsigned int BiosSelector;            /* PnP BIOS readable/writable selector */
```

## Description:

*Required.* This function will allow system software to set the system resource configuration for the specified System Device Node. The *Node* argument will contain the unique node number (handle) for the device that is to be set, and *devNodeBuffer* contains the pointer to the node data structure that specifies the new resource allocation request. The node data structure must completely describe the resource settings for the device. A node data structure that contains partial settings will result in the improper set up of the device. It cannot be assumed that any previous resource allocations will remain when this call is made. It is important to note that the resource descriptors that define the resource allocation must be specified in the same order as listed in the allocated resource configuration block for the system device node to be set. The allocated resource configuration block should be used as a template for setting the new resources for the device to ensure that the descriptors are specified in the correct format. In fact, the *devNodeBuffer* can be a copy of the fetched System Device Node with its allocated resource configuration block modified to reflect the desired new device configuration. Therefore, this function must be implemented to extract and use only the relevant new resource configuration information while ignoring all other extraneous node information. This function will not validate the resource settings or the checksum passed by the caller, and may not return an error code.

To disable a device, all resource descriptors in the allocated resource configuration block of the System Device Node must be set to zero. The resource attribute information field and the tag field are "Don't Care" and may be zeroed.

The *Control flag* provides a mechanism for allowing the system software to indicate whether the systemboard device configuration specified by this call is to take affect immediately or at the next boot. *Control* is defined as:

Bits 15:2: Reserved (0)
Bit 1: 0=Do not set the device configuration for the next boot.
1=Set the device configuration for the next boot (static configuration).
Bit 0: 0=Do not set the device configuration dynamically.
1=Set the device configuration right now (dynamic configuration).

If *Control flag* is 0, neither bit 0 nor bit 1 is set and this function should return BAD_PARAMETER. If both bits are set, then the system BIOS will attempt to set the configuration of the device right now (dynamic configuration), as well as set the device configuration for the next boot (static configuration). When both bits are set, it is possible that the NOT_SET_STATICALLY warning could be generated. This indicates that the device was configured dynamically, but could not be configured statically (See Appendix C, Error Codes).

The *BiosSelector* parameter enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode the caller must create a data segment descriptor using the 16-bit Protected Mode data segment base address specified in the Plug and Play Installation Check data structure, a limit of 64k, and the descriptor must be read/write capable. If this function is called from real mode, *BiosSelector* should be set to the Real Mode 16-bit data segment address as specified in the Plug and Play Installation Check structure. Refer to section 4.4 above for more information on the Plug and Play Installation Check Structure and the elements that make up the structure.

The function is available in real mode and 16-bit protected mode.

## Returns:

0 if successful - SUCCESS
!0 if an error (Bit 7 set) or a warning occurred - error code (The function return codes are described in Appendix C)
The FLAGS and registers will be preserved, except for AX which contains the return code.

**Example:**

The following example illustrates how the 'C' style call interface could be made from an assembly language module:

```
        .
        .
        .
push    Bios Selector
push    Control                         ; Control flag
push    segment/selector of devNodeBuffer   ; pointer to devNodeBuffer
push    offset of devNodeBuffer
push    Node                            ; node number - only low 8-bits used
push    SET_DEVICE_NODE                 ; Function 2
call    FAR PTR entryPoint
add     sp,12                           ; Clean up stack
cmp     ax,SUCCESS                      ; Function completed successfully?
jne     error                           ; No-handle error condition
        .
        .
        .
```

## 4.6 Event Notification Interface

Certain classes of systems may provide the capability for the addition or removal of system devices while the system unit is powered on, such as inserting a Notebook unit into a Docking Station. System BIOS support is necessary for providing Event Notification accessible to system software so that when devices are added or removed the system software will comprehend the use or release of system resources by those devices. Event Notification can be implemented as either a polled method or as asynchronous events. System software can check the *Control Word*, which is located in the *BIOS Plug and Play Header* structure, to determine the Event Notification method supported on the system. *Refer to the Plug and Play Installation Check section for more information on the BIOS Plug and Play Header and the Control Word.* The *Control Word* has bits defined that indicate the type of Event Notification. The BIOS Plug and Play Header structure also contains the *Event notification flag address*, which specifies the physical location of the *Event Flag* for polling. The *Event Flag* is the event polling location. When a system event occurs bit 0 of the *Event Flag* will be set to indicate a pending event. Therefore, if the method for Event Notification is through polling, system software should monitor the *Event Flag* to determine when a configuration event has occurred.
The asynchronous method of Event Notification allows system software to install an interrupt handler as a means for notification. The system BIOS will specify a system device node, which can be obtained from the *Get Node* runtime function, that will specify the requirements for handling asynchronous events. The system device node for asynchronous event management will be identified through the device identifier field in the device node data structure, and will specify the interrupt number and an I/O port address. This event system device node can be defined in one of two ways. First, the device node can follow the generic implementation in which the device identifier is PNP0C03 and the interrupt number and I/O address assigned are system specific. The only requirement with the generic implementation is that the I/O address bit used for detecting the source of the interrupt and clearing the interrupt line is bit 0. If bit 0 of this I/O address is set to 1, then the interrupt was generated due to a system event. The interrupt service routine should reset the interrupt line by clearing bit 0 at the specified I/O address. All other bits read from the I/O address should not be modified. The second way the event system device node can be defined is implementation specific where the system vendor must supply their own device identifier and whatever resources are required for servicing the event interrupt. This method will require a specific device driver associated with the device node identifier to support the event notification interface.

When the system software is notified of an event by either mechanism, it can then call the BIOS runtime function to get the event which will return a message specifying the type of event. These events are specific to the system and do not represent events that can occur on the various expansion busses, such as PCMCIA insertion and removal events. The following table describes the types of events that are reported through this BIOS interface:

| Event Identifier | Value | Description |
|---|---|---|
| ABOUT_TO_CHANGE_CONFIG | 0001h | This message provides the system with a mechanism whereby system software can obtain notification from the system BIOS when a change is about to be made to the system. This notification encompasses initiating a docking, or undocking, sequence. For systems that support this message, the docking sequence will be suspended until the system software issues a **Send_Message()** to the system BIOS with either an **OK** message indicating that it's OK to dock/undock the system, or an **ABORT** message that signals the BIOS to halt the event. *(Refer to Send Message function description below for more information.)* |
| DOCK_CHANGED | 0002h | This message indicates that new devices have either been successfully added or removed from the system, such as docking to, or undocking from, a docking station. This message will be used to indicate that a convenience base has been added/removed from the system. |
| SYSTEM_DEVICE_CHANGED | 0003h | This message indicates that removable ("pluggable") system devices have been removed or inserted into the base unit. |
| CONFIG_CHANGE_FAILED | 0004h | This message indicates that the system detected an error when attempting to add or remove devices to/from the system, such as attempting to dock to the docking station, or failing to successfully undock from the docking station. An error code will be returned in the return status for the *Get_Event* Plug and Play BIOS function if the system is able to determine the cause of the CONFIG_CHANGE_FAILED. Appendix C contains a complete list of return codes. |
| UNKNOWN_SYSTEM_EVENT | FFFFh | An unknown system event has occurred. The system BIOS is not able to determine the type of event. |
| OEM_DEFINED_EVENTS | 8000h thru FFFEh | OEM defined events allow OEM to define events specific to their system implementation. These events are only comprehended by the OEM. These events are identified by the upper bit of the event message being set (bit 7=1). |

To properly support event management, a PnP BIOS should implement the PNP_OS_ACTIVE and PNP_OS_INACTIVE messages, as well as their associated event timing requirements and PnP-OS-Active states as described in section 4.6.2.

### 4.6.1 Function 3 - Get Event

#### Synopsis:

```
int FAR (*entryPoint)(Function, Message, BiosSelector);
int Function;                          /* PnP BIOS Function 3 */
unsigned int FAR *Message;             /* Storage for the event message */
 unsigned int BiosSelector;            /* PnP BIOS readable/writable selector */
```

#### Description:

*Required for Dynamic Event Management.* This function will allow system software to retrieve a message specifying the type of event that has occurred on the system. This function is supported for either event notification by polling or for asynchronous event notification, if the system BIOS provides event notification. It is the responsibility of this function to clear the event flag when called if the event notification method implemented is through polling.

If a system event has occurred this call will return the appropriate event notification message in the memory location specified by the *Message* argument. *Message* will be set to one of the following event notification messages:

ABOUT_TO_CHANGE_CONFIG
DOCK_CHANGED
SYSTEM_DEVICE_CHANGED
CONFIG_CHANGE_FAILED
UNKNOWN_SYSTEM_EVENT
OEM_DEFINED_EVENT

The event notification messages are defined in the table at the start of Event Notification Interface section. If *Message* is CONFIG_CHANGE_FAILED and the system is able to determine the cause of the error, then the appropriate error should be returned in AX. This will allow system software the ability notify the user of the cause of the failure. Refer to Appendix C for a description of the error codes associated with the CONFIG_CHANGE_FAILED event message.

The *BiosSelector* parameter enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode the caller must create a data segment descriptor using the 16-bit Protected Mode data segment base address specified in the Plug and Play Installation Check data structure, a limit of 64k, and the descriptor must be read/write capable. If this function is called from real mode, *BiosSelector* should be set to the Real Mode 16-bit data segment address as specified in the Plug and Play Installation Check structure. Refer to section 4.4 above for more information on the Plug and Play Installation Check Structure and the elements that make up the structure.

This function is available in real mode and 16-bit protected mode.

#### Returns:

0 if successful - SUCCESS

!0 if an error (Bit 7 set) or a warning occurred or no pending events - error code (The function return codes are described in Appendix C)

The FLAGS and registers will be preserved, except for AX which contains the return code.

## Example:

The following example illustrates how the 'C' style call interface could be made from an assembly language module:

```
        .
        .
        .
push    Bios Selector
push    segment/selector of Message         ; pointer to Message
push    offset of Message
push    GET_EVENT                           ; Function 3
call    FAR PTR entryPoint
add     sp,8                                ; Clean up stack
cmp     ax,SUCCESS                          ; Function completed successfully?
jne     error                               ; No-handle error condition
        .
        .
        .
```

### 4.6.2 Function 4 - Send Message

## Synopsis:

```
int FAR (*entryPoint)(Function, Message, BiosSelector);
int Function;                   /* PnP BIOS Function 4 */
unsigned int Message;           /* Docking Message */
unsigned int BiosSelector;      /* PnP BIOS readable/writable selector */
```

## Description:

*Required for Dynamic Event Management.* This function will provide system software with a mechanism for interacting with the system while handling system events. There are three classes of messages that are supported by this interface: Response Messages, Control Messages, and OEM Defined Messages. The Response Messages are used as a means whereby the system BIOS will not proceed with a particular event until the system software provides a response instructing the system BIOS to continue or abort the processing of that event. Message values 0 through 3Fh are reserved for Response Messages. Control Messages provide system software with the ability to cause a particular event to happen. Message values 40h through 7Fh are reserved for Control Messages. OEM Defined Messages are specific to the OEM's system implementations and are only understood by the OEM. Message values 8000h through FFFFh identify OEM Defined Messages. The following table describes the event messages that system software can send to the system BIOS, where *Message* has one of the following meanings:

**Response Messages 00h through 3Fh:**

| Message Identifier | Value | Description |
|---|---|---|
| OK | 00h | Instructs the system to continue with the sequence which initiated the event. This message is only valid when the Get Event function has returned one of the ABOUT_TO_XXXXX events. When the system software is notified with an ABOUT_TO_XXXXX message, the appropriate actions will not take place until the Send Message BIOS Function is called with OK. |
| ABORT | 01h | Abort the action which initiated the ABOUT_TO_XXXXX event. This message instructs the system BIOS to prevent the event from occurring. For instance, if the event is an undocking sequence, then the system will not be allowed to undock. It is assumed that it is the responsibility of the system software to communicate to the user the reason for not allowing the system to carry out the action for the event. This message is only valid when Get Event has returned one of the ABOUT_TO_XXXXX messages. |

**Control Messages 40h through 7Fh:**

| Message Identifier | Value | Description |
|---|---|---|
| UNDOCK_DEFAULT_ACTION | 40h | This message provides a mechanism for system software to soft eject the system and instructs the system BIOS to take the default action when ejecting the system. |
| POWER_OFF | 41h | This message instructs the system BIOS to power off the system. It is assumed that the system software will perform the necessary actions to shut the system down before sending this message. |
| PNP_OS_ACTIVE | 42h | This message allows the PnP BIOS to track whether a PnP OS is active and defines event timing. The PnP BIOS may default to either a PnP-OS-Inactive or PnP-OS-Active state as needed. However, upon initial OS load, a PnP OS will register with the PnP BIOS by sending the PNP_OS_ACTIVE message to the PnP BIOS. When the PNP_OS_ACTIVE message is received, the PnP BIOS will operate in the PnP-OS-Active state. In this state, the PnP BIOS will wait forever after signaling a system event. This will allow the PnP OS to execute a *Plug and Play BIOS Function Get Event* call and handle the event (See Section 4.6). Although a PnP BIOS is not required to support the PNP_OS_ACTIVE message, support is recommended in systems that generate events. If this message is unsupported, then MESSAGE_NOT _SUPPORTED should be returned. |
| PNP_OS_INACTIVE | 43h | This message complements the PNP_OS_ACTIVE message. A PnP OS will send the PNP_OS_INACTIVE message to the PnP BIOS upon OS termination. When the PNP_OS_INACTIVE message is received, the PnP BIOS will operate in the PnP-OS-Inactive state. In this state, no PnP event timing constraints exist. The PnP BIOS does not have to wait for the PnP OS to execute a *Plug and Play BIOS Function Get Event* call (See Section 4.6) and can handle event timing in the manner it best determines. Although a PnP BIOS is not required to support the PNP_OS_INACTIVE message, support is recommended in systems that generate events. If this message is unsupported, then MESSAGE_NOT _SUPPORTED should be returned. |

**OEM Defined Messages 8000h through FFFFh:**

| Message Identifier | Value | Description |
|---|---|---|

| | | |
|---|---|---|
| OEM_DEFINED_MESSAGES | 8000h thru FFFFh | This message allows OEMs to define messages specific to their system implementation. These messages are only comprehended by the OEM. These messages are identified by the upper bit of the message. |

If the system BIOS does not support one of the specified messages, this function will return MESSAGE_NOT_SUPPORTED.

The *BiosSelector* parameter enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit Protected Mode data segment base address specified in the Plug and Play Installation Check data structure, a limit of 64KB, and the descriptor must be read/write capable. If this function is called from real mode, *BiosSelector* should be set to the Real Mode 16-bit data segment address as specified in the Plug and Play Installation Check structure. Refer to section 4.4 above for more information on the Plug and Play Installation Check Structure and the elements that make up the structure.

This function is available in real mode and 16-bit protected mode.

## Returns:

0 if successful - SUCCESS

!0 if an error (Bit 7 set) or a warning occurred - error code (The function return codes are described in Appendix C)

The FLAGS and registers will be preserved, except for AX which contains the return code.

## Example:

The following example illustrates how the 'C' style call interface could be made from an assembly language module:

```
        .
        .
        .
push    Bios Selector
push    Message                         ; Message
push    SEND_MSG                        ; Function 4
call    FAR PTR entryPoint
add     sp,6                            ; Clean up stack
cmp     ax,SUCCESS                      ; Function completed successfully?
jne     error                           ; No-handle error condition
        .
        .
        .
```

### 4.6.3 Function 5 - Get Docking Station Information

## Synopsis:

*int FAR (*entryPoint)(Function, DockingStationInfo, BiosSelector);*
*int Function;* */* PnP BIOS Function 5 */*
*unsigned char FAR *DockingStationInfo;* */* Pointer to docking station info structure */*
*unsigned int BiosSelector;* */* PnP BIOS readable/writable selector */*

## Description:

*Required for Dynamic Event Management.* This function will allow system software to get information which specifies the type of docking device, either expansion or convenience base, the system is connected to, as well as the capabilities of the docking device. The docking station information will be returned in the memory buffer pointed to by *DockingStationInfo* in the following format:

| Field | Offset | Length | Value |
|---|---|---|---|
| Docking station location identifier | 00h | DWORD | Varies |
| Serial number | 04h | DWORD | Varies |
| Capabilities | 08h | WORD | Varies |

***Docking station location identifier:***
This field is the docking device location identifier. The identifier should follow the EISA device identifier format. The docking device location identifier will allow system software to differentiate between the types of docking stations and convenience bases that the base system unit can be connected to. This enables the system software to better determine the various docked and undocked configuration states. *LocationId* will be set to UNKNOWN_DOCKING_IDENTIFIER (0xFFFFFFFF) for docking stations and/or convenience bases that do not have a product identifier.

***Serial number:***
*SerialNum* is not required; however, if the docking station does not have a serial number, then 0 should be returned in this parameter.

***Capabilities:***
The Docking Capabilities bit field is defined as follows:

Bits 15:3 Reserved (0)
Bit 2:1 - 00=System should be powered off to dock or undock (Cold Docking)
01=System supports Warm Docking/Undocking, system must be in suspend
10=System supports Hot Docking/Undocking, not required to be in suspend
11=Reserved
Bit 0 - 0=Docking station does not provide support for controlling the docking/undocking sequence (Surprise Style).
1=Docking station provides support for controlling the docking/undocking sequence (VCR Style).

If the system supports docking and is unable to determine the docking station capabilities, this function will return UNABLE_TO_DETERMINE_DOCK_CAPABILITIES. All other relevant information, such as the docking station identifier, will be returned in the data structure.
If the system does not support docking, this function will return FUNCTION_NOT_SUPPORTED. If the system supports docking, but is not currently docked, this function will return SYSTEM_NOT_DOCKED and will not return any information about a docking station.
The *BiosSelector* parameter enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit Protected Mode data segment base address specified in the Plug and Play Installation Check data structure, a limit of 64KB, and the descriptor must be read/write capable If this function is called from real mode, *BiosSelector* should be set to the Real Mode 16-bit data segment address as specified in the Plug and Play Installation Check structure. Refer to

section 4.4 above for more information on the Plug and Play Installation Check Structure and the elements that make up the structure.
The function is available in real mode and 16-bit protected mode.

## Returns:

0 if successful - SUCCESS
!0 if an error (Bit 7 set) or a warning occurred or the system is not currently docked (The function return codes are described in Appendix C)
The FLAGS and registers will be preserved, except for AX which contains the return code.

## Example:

The following example illustrates how the 'C' style call interface could be made from an assembly language module:

```
        .
        .
        .
push    Bios Selector
push    segment/selector of DockingStationInfo      ; pointer to docking station info data structure
push    offset of DockingStationInfo
push    GET_DOCK_INFO                               ; Function 5
call    FAR PTR entryPoint
add     sp,8                                        ; Clean up stack
cmp     ax,SUCCESS                                  ; Function completed successfully?
jne     error                                       ; No-handle error condition
        .
        .
```

### 4.6.4 Function 6 - Reserved

This function has been reserved for future implementations and should return FUNCTION_NOT_SUPPORTED.

### 4.6.5 Function 7 - Reserved

This function has been reserved for future implementations and should return FUNCTION_NOT_SUPPORTED.

### 4.6.6 Function 8 - Reserved

This function has been reserved for future implementations and should return FUNCTION_NOT_SUPPORTED.

# 4.7 Extended Configuration Services

This section describes the optional extended services provided by the System BIOS on Plug and Play platforms.

The extended configuration services are a mechanism whereby the system software may specify the system resources assigned to devices that have been installed in the system. This information will be maintained by the BIOS in some form of nonvolatile storage. Depending upon the amount of nonvolatile storage available to store system configuration information, one can either store detailed configuration information for all devices or limit the information to a description of the summary resource usage by the static ISA devices in the system. In both cases, this information is to help the BIOS configure boot devices during the Power On Self Test (POST) phase. See ***Section 2.1.4*** in the system POST area for a more complete description of the POST process.

## Get & Set Statically Allocated Resources

Functions 9 and Ah allow the OS to effectively reserve resources allocated by legacy cards in a system. This provides a resource usage map for the BIOS to use to avoid resource conflicts when allocating resources to other devices. Only summary resource usage information by the legacy ISA cards must be stored in nonvolatile storage. This information describes the cumulative usage of system resources by all legacy ISA cards but does not identify the specific resources used by each card. The POST configuration software will use this information to avoid resource conflicts when configuring boot devices. This solution can be implemented with minimum NVRAM; however, it does afford less control over the configuration. The example in section 4.7.1 describes how Plug and Play ISA resource descriptor information can be stored compactly. The storage structure definition is left completely up to the OEM.

The operating system should call function 9 to determine if the platform Plug and Play interface supports the ISA resource descriptors or not. If this call returns without an error, it can be assumed that the platform is storing the ISA resource descriptor information in a proprietary bit map format. If function calls 9 or 0Ah return the USE_ESCD_SUPPORT error message, then the caller can assume that this platform supports the ESCD method of data storage.

## Read & Write Extended System Configuration Data (ESCD)

The ESCD data storage method allows OEMs to differentiate a platform with additional Plug and Play features. Since the data format stores information about which devices are using what resources, it is possible to maintain an image of the Last Working Configuration of all know devices. Additionally, system software can modify the ESCD at runtime and affect the configuration of devices for the next boot. This allows bootable devices to be enabled/disabled and other devices to be locked into specific configurations. The ESCD also provides detailed configuration information about static devices allowing the POST configuration software to avoid conflicts with these cards. In general, the ESCD allows the Plug and Play system BIOS to more fully configure the system at power up; this is important for platforms that must support non-Plug and Play operating systems.

The ESCD format describes every device in the system so storage requirements are much larger. A typical platform requires 2-4KB of NVRAM. The Plug and Play interface can support a function call that allows the caller to Get NVRAM size attributes, and it supports two other functions that provide Read/Write access to the Extended System Configuration Data where it is stored in the NVRAM.

The operating system should call function 9 to determine which data storage format this platform's Plug and Play interface supports. If the function 9 call returns without an error, it can be assumed that the platform is storing the ISA resource descriptor information in a proprietary bit map format. If function calls 9 or 0Ah return the USE_ESCD_SUPPORT error message, then the caller can assume that this platform supports the ESCD method of data storage.

More detailed and current information about the ESCD definition and format specification can be found in the *ESCD Specification.*

### 4.7.1 Function 9 - Set Statically Allocated Resource Information

#### Synopsis:

```
int FAR (*entryPoint)(Function, ResourceBlock, BiosSelector);
int Function;                       /* PnP BIOS Function 9 */
unsigned char FAR *ResourceBlock;   /* Block of statically allocated resources */
unsigned int BiosSelector;          /* PnP BIOS readable/writable selector */
```

#### Description:

*Optional.* This function will allow system software to report the system resources that are being utilized by the static ISA devices installed in the system. The system software must pass a complete list of system resources used by ALL of the legacy ISA devices that are not located on the system board. Therefore, any time a legacy ISA device is added or removed from the system, the system software must construct a new resource map and pass the information to the system BIOS by making this function call. This information is important to the Plug and Play BIOS POST functionality for achieving the ability to bootstrap the operating system from a Plug and Play boot device by allowing the Plug and Play BIOS to configure the boot device around the legacy ISA devices. The resources allocated to the legacy ISA devices in the system are reported in the *ResourceBlock* parameter. The format of the data contained in the block follows the format defined in the *Plug and Play ISA Specification* under the section labeled *Plug and Play Resources.* This data is provided as a series of data structures with each structure having a unique tag or identifier. The resource descriptors supported by this function are the descriptors that describe IRQ, DMA, I/O addresses, and memory resources. The resource information specified in this block must be terminated with an *END_TAG* resource descriptor.

The *BiosSelector* parameter enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit Protected Mode data segment base address specified in the Plug and Play Installation Check data structure, a limit of 64KB, and the descriptor must be read/write capable. If this function is called from real mode, *BiosSelector* should be set to the Real Mode 16-bit data segment address as specified in the Plug and Play Installation Check structure. Refer to section 4.4 above for more information on the Plug and Play Installation Check Structure and the elements that make up the structure.

If this function returns USE_ESCD_SUPPORT, then reporting resources allocated to devices to the system BIOS must be handled through the interface defined by the *ESCD Specification* (see sections 4.7.4 - 4.7.6, functions 41h, 42h and 43h.).

This function is available in real mode and 16-bit protected mode.

#### Returns:

0 if successful - SUCCESS

!0 if an error (Bit 7 set) or a warning occurred or no pending events - error code (The function return codes are described in Appendix C)

The FLAGS and registers will be preserved, except for AX which contains the return code.

#### Example:

The following example illustrates how the 'C' style call interface could be made from an assembly language module:

```
        .
        .
        .
push    Bios Selector
push    segment/selector of the Resource Block      ; Pointer to the data structure of isa resources
push    offset of Resource Block
push    SET_STATICALLY_ALLOCATED_RESOURCES          ; Function 9
call    FAR PTR entryPoint
add     sp,8                                        ; Clean up stack
cmp     ax,SUCCESS                                  ; Function completed successfully?
jne     error                                       ; No-handle error condition
        .
```

.
.

A BIOS implementor is only required to follow the interface described by this function. The format of the data passed by the system software must follow the Plug and Play ISA resource descriptor definition. How the statically allocated resource information is actually stored is left up to the BIOS implementor. An example of how the information could be stored more compactly than the Plug and Play ISA resource descriptors is as follows:

| | |
|---|---|
| IRQ | 2 Bytes - Bits set indicate IRQ used by unconfigurable ISA device |
| DMA | 1 Byte - Bits set indicate DMA used by unconfigurable ISA device |
| I/O | 24 Bytes - Bits set indicate I/O addresses used (100h-3ffh). Assumes 4 ports used per I/O address bit set |
| Memory 640k to 1Mg | 3 Bytes - Represented in 16k blocks |
| Memory 1Mg to 16Mg | 2 Bytes - Represented in 1Mg increments. |

Storing the information this way would allow the system resources used by unconfigurable ISA devices to be contained in 32 bytes.

***Note: this is only an example. It is completely up to the BIOS vendor to choose an appropriate format for storing the data, which means it could possibly be stored in less than 32 bytes or require more than 32 bytes.***

### 4.7.2 Function 0Ah - Get Statically Allocated Resource Information

## Synopsis:

```
int FAR (*entryPoint)(Function, ResourceBlock, BiosSelector);
int Function;                       /* PnP BIOS Function 0Ah */
unsigned char FAR *ResourceBlock;   /* Block of resources statically allocated to devices */
unsigned int BiosSelector;          /* PnP BIOS readable/writable selector */
```

## Description:

*Optional.* This function will return the system resources that are being utilized by the legacy ISA devices that are installed in the system. These system resources are the resources that have been reported to the system BIOS through the *Set Allocated ISA Resource Info* function. The resources allocated to the legacy ISA devices in the system are reported in the *ResourceBlock* parameter. It is important to note that the information returned represents the resource usage rounded up to the nearest granularity range supported by the system BIOS and not the actual resources used by the legacy ISA devices in the system. It is recommended that the system software keep track of the system resources used by legacy ISA cards in order to account for the exact system resources usage of the legacy ISA cards installed in the system. The format of the data contained in the block follows the format defined in the *Plug and Play ISA Specification* under the section labeled *Plug and Play Resources.* This data is provided as a series of data structures with each structure having a unique tag or identifier. The *ResourceBlock* must be a minimum of 2 Kbytes to ensure that there is adequate space for the system BIOS to return the legacy ISA resource information.

The *BiosSelector* parameter enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode the caller must create a data segment descriptor using the 16-bit Protected Mode data segment base address specified in the Plug and Play Installation Check data structure, a limit of 64k, and the descriptor must be read/write capable. If this function is called from real mode, *BiosSelector* should be set to the Real Mode 16-bit data segment address as specified in the Plug and Play Installation Check structure. Refer to section 4.4 above for more information on the Plug and Play Installation Check Structure and the elements that make up the structure.

If this function returns USE_ESCD_SUPPORT, then accessing the information describing the resources allocated to devices to the system BIOS must be handled through the interface defined by the *ESCD Specification.* Refer to the *ESCD Specification* for a complete description of the interfaces to support the ESCD as well as the format of the ESCD.

This function is available in real mode and 16-bit protected mode.

## Returns:

0 if successful - SUCCESS

!0 if an error (Bit 7 set) or a warning occurred or no pending events - error code (The function return codes are described in Appendix C)

The FLAGS and registers will be preserved, except for AX which contains the return code.

## Example:

The following example illustrates how the 'C' style call interface could be made from an assembly language module:

```
        .
push    Bios Selector
push    segment/selector of the Resource Block      ; Pointer to the data struct of isa resources
push    offset of Resource Block
push    GET_STATICALLY_ALLOCATED_RESOURCES          ; Function 0Ah
call    FAR PTR entryPoint
add     sp,8                                        ; Clean up stack
cmp     ax,SUCCESS                                  ; Function completed successfully?
jne     error                                       ; No-handle error condition
        .
```

### 4.7.3 Function 40h - Get Plug & Play ISA Configuration Structure

## Synopsis:

```
int FAR (*entryPoint)(Function, Configuration, BiosSelector);
int Function;                   /* PnP BIOS Function 40h */
unsigned char FAR *Configuration;   /* Address of caller's config. structure buffer*/
unsigned int BiosSelector;      /* PnP BIOS readable/writable selector */
```

## Description:

*Required.* This function is used to get the Plug and Play ISA Configuration structure. The Plug and Play ISA Configuration data structure contains configuration information specific to ISA Plug and Play support. This function will copy the data structure to the caller's memory buffer specified by *Configuration*. A system without any ISA bus capabilities will return the FUNCTION_NOT_SUPPORTED error code. When the ISA bus is present, the fields in this data structure will be set with the appropriate values. If the system BIOS did not identify any Plug and Play ISA cards in the system during POST, then the *Total number of Card Select Numbers* field will be zero and the value in the *ISA Read Data Port* field is invalid and must not be used by system software.
On systems with a dynamic ISA bus, like portables, function 40h will be more flexible. When an ISA bus is present, the information returned by function 40h will always be valid after a cold boot. On a cold boot with no ISA bus present, function 40h will return zeros. After an ISA warm/hot dock, the function 40h information will also be valid, if the plug and play BIOS isolates and enumerates the plug and play adapter cards before returning control to the plug and play operating system. If the BIOS does not re-enumerate after an ISA warm/hot dock event, then the information returned by function 40h will be zeros. After an ISA undock event, this information will also be zeros.
The format of the Plug and Play ISA Configuration structure is defined as follows:

| Field | Offset | Length | Value |
|---|---|---|---|
| Structure Revision | 00h | BYTE | 01 |
| Total number of Card Select Numbers (CSNs) assigned | 01h | BYTE | Varies |
| ISA Read Data Port | 02h | WORD | Varies |
| Reserved | 04h | WORD | 0 |

***Structure Revision:***
This is an ordinal value that indicates the revision number of this structure only and does not imply a level of compliance with the Plug and Play BIOS version.
***Total number of Card Select Numbers:***
This field specifies the total number of CSNs assigned to ISA Plug and Play cards by the system BIOS during the Power-On Self Test (POST).

***ISA Read Data Port:***
The ISA Read Data Port is used to read information from the Plug and Play registers. The value represented here is the I/O port that was determined by the system BIOS to not conflict with another ISA I/O port. *Refer to the ISA Plug and Play Specification for more information on the ISA Read Data Port.*
The *BiosSelector* parameter enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode, the caller must create a data segment descriptor using the 16-bit Protected Mode data segment base address specified in the Plug and Play Installation Check data structure, a limit of 64KB, and the descriptor must be read/write capable. If this function is called from real mode, *BiosSelector* should be set to the Real Mode 16-bit data segment address as specified in the Plug and Play Installation Check structure. Refer to section 4.4 above for more information on the Plug and Play Installation Check Structure and the elements that make up the structure.
This function is available in real mode and 16-bit protected mode.

## Returns:

0 if successful - SUCCESS
!0 if an error (Bit 7 set) or a warning occurred - error code (The function return codes are described in Appendix C)
The FLAGS and registers will be preserved, except for AX which contains the return code.

## Example:

The following example illustrates how the 'C' style call interface could be made from an assembly language module:

```
push    Bios Selector
push    segment/selector of Config. structure buffer ; pointer to configuration data buffer
push    offset of Configuration structure buffer
push    GET_ISA_CONFIG_STRUC            ; Function 40h
call    FAR PTR entryPoint
add     sp,8                            ; Clean up stack
cmp     ax,SUCCESS                      ; Function completed successfully?
jne     error                           ; No-handle error condition
        .
        .
```

### 4.7.4 Function 41h - Get Extended System Configuration Data (ESCD) Info

## Synopsis:

```
int FAR (*entryPoint)(Function, MinESCDWriteSize, ESCDSize, NVStorageBase,
                      BiosSelector);
int Function;                           /* PnP BIOS Function 041h */

unsiged int FAR *MinESCDWriteSize;      /* Minimum buffer size in bytes for writing to NVS */
unsigned int FAR *ESCDSize;             /* Size allocated for the ESCD... */
                                        /* ...within the nonvolatile storage block */
unsigned long FAR *NVStorageBase;       /* 32-bit physical base address for.*/
                                        /* .mem mapped nonvolatile storage media */
unsigned int BiosSelector;              /* PnP BIOS readable/writable selector */
```

## Description:

*Optional*. This function provides information about the nonvolatile storage on the system that contains the Extended System Configuration Data (ESCD). It returns the size, in bytes, of the minimum buffer required for writing to NVS in *MinESCDWriteSize*, the maximum size, in bytes, of the block within the nonvolatile storage area allocated specifically to the ESCD in *ESCDSize*, and if the nonvolatile storage is memory mapped, the 32-bit absolute physical base address will be returned in *NVStorageBase*. The physical base address of the memory mapped nonvolatile storage will allow the caller to construct a 16-bit

data segment descriptor with a limit of 64K and read/write access. This will enable the Plug and Play system BIOS to read and write the memory mapped nonvolatile storage in a protected mode environment. If the nonvolatile storage is not memory mapped the value returned in *NVStorageBase* should be 0. It is assumed that the size of the nonvolatile storage that contains the ESCD will not exceed 32K bytes.
Refer to the *ESCD Specification* for a complete description of the interfaces to support the ESCD as well as the format of the ESCD.

### 4.7.5 Function 42h - Read Extended System Configuration Data (ESCD)

#### Synopsis:

```
int FAR (*entryPoint)(Function, ESCDBuffer, ESCDSelector, BiosSelector)
int Function;                   /* PnP BIOS Function 042h */
char FAR *ESCDBuffer;           /* Addr of caller's buffer for storing ESCD */
unsigned int ESCDSelector;      /* ESCD readable/writable selector */
unsigned int BiosSelector;      /* PnP BIOS readable/writable selector */
```

#### Description:

*Optional*. This function is used to read the ESCD data from nonvolatile storage on the system into the buffer specified by *ESCDBuffer*. The entire ESCD will be placed into the buffer. It is the responsibility of the caller to ensure that the buffer is large enough to store the entire ESCD. The caller should use the output from Function 41 (the *ESCDSize* field) when calculating the size of the *ESCDBuffer*. The system BIOS will return the entire ESCD, including information about system board devices. The system board device configuration information will be contained in the slot 0 portion of the ESCD. The caller can determine the size of the data in the ESCD from the *ESCD Configuration Header Structure*. In protected mode, the *ESCDSelector* has base = *NVStorageBase* and limit of at least *NVStorageSize*. In real mode, the *ESCDSelector* is a segment that points to *NVStorageBase*.
Refer to the *ESCD Specification* for a complete description of the interfaces to support the ESCD as well as the format of the ESCD.

### 4.7.6 Function 43h - Write Extended System Configuration Data (ESCD)

#### Synopsis:

```
int FAR (*entryPoint)(Function, ESCDBuffer, ESCDSelector, BiosSelector);
int Function;                   /* PnP BIOS Function 043h */
char FAR *ESCDBuffer;           /* Buffer containing complete ESCD to write... */
                                /* ...to nonvolatile storage */
unsigned int ESCDSelector;      /* ESCD readable/writable selector */
unsigned int BiosSelector;      /* PnP BIOS readable/writable selector */
```

#### Description:

*Optional*. This function will write the Extended Static Configuration Data (ESCD) contained in the *ESCDBuffer* to nonvolatile storage on the system. The data contained in the caller's buffer must contain a complete block of ESCD structures describing the configuration information for devices on the system. The caller should use the output from Function 41 (the MinESCDWriteSize field) when calculating the size of the *ESCDBuffer*. The system BIOS can determine the size of the data in the ESCD structure from the *ESCD Configuration Header Structure* within the caller's ESCD buffer.
Refer to the *ESCD Specification* for a complete description of the interfaces to support the ESCD as well as the format of the ESCD.

## 4.8 Power Management Services

The following subsections describe the Plug and Play support for allowing interaction with Advanced Power Management (APM) 1.1 and greater.

### 4.8.1 Function 0Bh - Get APM ID Table

**Synopsis:**

```
int FAR (*entryPoint)(Function, Bufsize, ApmIdTable, BiosSelector);
int Function;                   /* PnP BIOS Function 0Bh */
unsigned int FAR *BufSize;      /* Size of buffer to copy APM ID table to */
unsigned char FAR *ApmIdTable;  /* Address of caller's buffer for the table */
unsigned int BiosSelector;      /* PnP BIOS readable/writable selector */
```

**Description:**

*Required for Power Management.* This function will copy the table of APM 1.1 (or greater) device identifier to Plug and Play device identifier mappings to the buffer specified by the caller. This allows the operating system to use the APM interface to perform power management on individual devices controlled by the system BIOS. If *BufSize* indicates that the buffer is not large enough to contain the entire table, the system BIOS will return BUFFER_TOO_SMALL and the size of the buffer required to contain the entire table will be returned in the caller's *BufSize* parameter. Therefore, the caller can call this function with *BufSize* equal to 0 to determine the size of the buffer it needs to allocate for the APM identifier table. The *apmIDTable* argument contains the pointer to the caller's memory buffer. If the buffer is large enough, on return *apmIDTable* will contain the APM identifier table. Each entry in the table will be specified in the following format:

| Field | Length | Value |
|---|---|---|
| Device identifier | DWORD | Varies |
| APM 1.1 identifier (version 1.1 or greater) | WORD | Varies |

***Device Identifier:***

This field is the Plug and Play device identifier. The Logical Device ID provides a mechanism for uniquely identifying multiple logical devices embedded in a single physical board. The format of the logical device ID is composed of three character compressed ASCII EISA ID and a manufacturer specific device code.

***APM identifier:***

This element specifies the corresponding APM device identifier.

An APM identifier table with multiple entries would be described as follows:

| Field |
|---|
| Device identifier #1 |
| APM identifier #1 |
| Device identifier #2 |
| APM identifier #2 |
| : |
| : |
| Device identifier #n |
| APM identifier #n |

This call supports APM version 1.1 or greater. The APM interface INT 2Fh supports a get version call. The *BiosSelector* parameter enables the system BIOS, if necessary, to update system variables that are contained in the system BIOS memory space. If this function is called from protected mode the caller must create a data segment descriptor using the 16-bit Protected Mode data segment base address

specified in the Plug and Play Installation Check data structure, a limit of 64K, and the descriptor must be read/write capable. If this function is called from real mode, *BiosSelector* should be set to the Real Mode 16-bit data segment address as specified in the Plug and Play Installation Check structure. Refer to section 4.4 above for more information on the Plug and Play Installation Check Structure and the elements that make up the structure.
This function is available in real mode and 16-bit protected mode.
Example scenario: An operating system provides a device driver level-interface to both the Plug and Play BIOS as well as the APM 1.1 (or greater) interface. An OEM or third party wishes to write a Plug and Play device driver for a device built into the system in order to provide enhancements available through operating-system services. However, he also wishes to power manage the device using support already available in the machine's APM 1.1 implementation. This function provides a means for the device driver to determine which Plug and Play identifiers have corresponding power management support through an APM 1.1 device identifier.

## Returns:

0 if successful - SUCCESS
!0 if an error (Bit 7 set) or a warning occurred - error code (The function return codes are described in Appendix C)
The FLAGS and registers will be preserved, except for AX which contains the return code.

## Example:

The following example illustrates how the 'C' style call interface could be made from an assembly language module:

```
        .
        .
        .
push    Bios Selector
push    segment/selector of APM Id table        ; pointer to APM Id table buffer
push    offset of APM Id table
push    segment/selector of table buffer size   ; pointer to APM Id table buffer size
push    offset of APM Id table buffer size
push    GET_APM_TABLE                           ; Function 0Bh
call    FAR PTR entryPoint
add     sp,12                                   ; Clean up stack
cmp     ax,SUCCESS                              ; Function completed successfully?
jne     error                                   ; No-handle error condition
        .
        .
        .
```

# Appendix A: Generic Option ROM Headers

## Generic Option ROM Header expansion

(Offsets are all based from the beginning of the Header)

| Offset | Length | Value | Description | |
|---|---|---|---|---|
| 0h | DWORD | $??? (ASCII) | Signature | Generic |
| 04h | BYTE | Varies | Structure Revision | Generic |
| 05h | BYTE | Varies | Length (in 16 byte increments) | Generic |
| 06h | WORD | Varies | Offset of next Header (0000 if none) | Generic |
| 08h | BYTE | 0FFFFh | Reserved | Generic |
| 09h | BYTE | Varies | Checksum | Generic |
| 10h | Varies | Varies | Specific Header Type Data | Specific |

**Signature** - All Expansion Headers will contain a unique expansion header identifier. Each different Expansion Header will have its own unique signature. Software that wishes to make use of any given Expansion Header simply traverses the linked list of Generic Expansion Headers until the Expansion Header with the desired signature is found, or the end of the list is encountered.
Example: The Plug and Play expansion header's identifier is the ASCII string "$PnP" or hex 24 50 6E 50h.
**Structure Revision** - This is an ordinal value that indicates the revision number of this structure only and does not imply a level of compliance with the Plug and Play BIOS version.
**Length** - Length of the entire Expansion Header expressed in sixteen byte blocks. The length count starts at the Signature field.

**Offset of Next Header** - This location contains a link to then next expansion ROM header in this Option ROM. If there are no other expansion ROM headers then this field will have a value of 0h.
**Reserved** - Reserved for Expansion
**Checksum** - Each Expansion Header is checksummed individually. This allows the software that wishes to make use of an expansion header the ability to determine if the expansion header is valid.
The system software can determine if the expansion header is valid by performing a **Checksum** operation. The method for validating the checksum is to add up *Length* bytes, including the *Checksum* field, into an 8-bit value. A resulting sum of zero indicates a valid checksum operation.
**Specific Header data** - This area is used by the specific device header and is defined uniquely for each Expansion Header.

# Appendix B: Device Driver Initialization Model

*Please Note: The Device Driver Initialization Model (DDIM) is provided as an extension of the current option ROM model. Current bus devices cannot be guaranteed that the systems in which they are installed will support DDIM. Therefore, current bus device Option ROMs (ISA, EISA, MCA, PCMCIA) must support the standard initialization model, and may optionally support the DDIM. The Option ROM may determine if it is being initialized using a DDIM by attempting to write to, and read back from his data space. If the Option ROM can successfully write to its data space, then it should support a DDIM initialization. Otherwise, it must perform a standard initialization.*
*As of this writing, the PCI architecture is the only architecture wherein Option ROMs are guaranteed support for DDIM.*
In an effort to reduce the amount of UMB (Upper Memory Block) space consumed by add-in Option ROMs, and to more efficiently use the available UMB space, Plug and Play Option ROMs should support the Device Driver Initialization Model (DDIM).
Under this model, all Option ROMs installed in a Plug and Play system which indicate that they support DDIM will be copied into RAM by the System BIOS. The System BIOS will then execute a FAR CALL into the device's initialization vector.
Devices which support DDIM may then initialize themselves, update their RAM image with static Data (if necessary), and then discard the initialization code, by updating the length byte at offset 3h and recalculating their checksum. The System BIOS will then initialize the next DDIM ROM by copying it to RAM on the next 2 KB boundary following the end of the most recently initialized DDIM ROM (or in the next available UMB which is large enough to contain both the Runtime and Initialization code of the DDIM ROM).
Once all DDIM Option ROMs have been initialized, the System BIOS will Write Protect the RAM images and proceed with the boot process.

## Flow:

System BIOS copies the DDIM Option ROM Copy to RAM
System BIOS executes a FAR CALL to Initialization Vector
Option ROM initializes the device
Option ROM updates any static data structures
Option ROM updates the ROM Length and Check sum
Option ROM returns to the System BIOS with Return Status
System BIOS Write Protects RAM image of ROM.

## Advantages:

* Provides more efficient use of Upper Memory Blocks. Initialization code may be discarded.

* Provides a seamless mechanism whereby Option ROMs may be copied to RAM

* Provides Option ROMs with a means of storing Static Data Structures built at boot time.

* Allows board vendors to use lower performance ROM devices (on buses that are guaranteed to support this architecture - PCI).

# Appendix C: Return Codes

The following table represents the return codes for the BIOS functions.
Bit 7 set indicates an error has occurred.

**Success Codes 00h:**

| Return Code | Value | Description |
|---|---|---|
| SUCCESS | 00h | Function completed successfully |

**Warning Codes 01h through 7Fh:**

| Return Code | Value | Description |
|---|---|---|
| Reserved | 01h | |
| NOT_SET_STATICALLY | 7Fh | Warning that indicates a device could not be configured statically, but was successfully configured dynamically. This return code is used only when function 02h is requested to set a device both statically and dynamically. |

**Error Codes 81h through FFh:**

| Return Code | Value | Description |
|---|---|---|
| UNKNOWN_FUNCTION | 81h | Unknown, or invalid, function number passed |
| FUNCTION_NOT_SUPPORTED | 82h | The function is not supported on this system. |
| INVALID_HANDLE | 83h | Device node number/handle passed is invalid or out of range. |
| BAD_PARAMETER | 84h | Function detected invalid resource descriptors or resource descriptors were specified out of order. |
| SET_FAILED | 85h | Set Device Node function failed. |
| EVENTS_NOT_PENDING | 86h | There are no events pending. |
| SYSTEM_NOT_DOCKED | 87h | The system is currently not docked. |
| NO_ISA_PNP_CARDS | 88h | Indicates that no ISA Plug and Play cards are installed in the system. |
| UNABLE_TO_DETERMINE_DOCK_ CAPABILITIES | 89h | Indicates that the system was not able to determine the capabilities of the docking station. |
| CONFIG_CHANGE_FAILED_NO_ BATTERY | 8Ah | The system failed the undocking sequence because it detected that the system unit did not have a battery. |
| CONFIG_CHANGE_FAILED_ RESOURCE_CONFLICT | 8Bh | The system failed to successfully dock because it detected a resource conflict with one of the primary boot devices; such as Input, Output, or the IPL device. |
| BUFFER_TOO_SMALL | 8Ch | The memory buffer passed in by the caller was not large enough to hold the data to be returned by the system BIOS. |
| USE_ESCD_SUPPORT | 8Dh | This return code is used by functions 09h and 0Ah to instruct the caller that reporting resources explicitly assigned to devices in the system to the system BIOS must be handled through the interfaces defined by the *ESCD Specification.* |
| MESSAGE_NOT_SUPPORTED | 8Eh | This return code indicates the message passed to the system BIOS through function 04h, Send Message, is not supported on the system. |

| | | |
|---|---|---|
| HARDWARE_ERROR | 8Fh | This return code indicates that the system BIOS detected a hardware failure. |